

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Introduction

Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- Web Services provide a standard way to make functionality available to client-side developers, independent of programming language and other implementation details
- Web Services are used in many applications at Fidelity
- RESTful web services provide a lightweight means of providing data in response to requests sent via HTTP
- This will be very useful in web applications where JavaScript will send a request to a RESTful service and will display the data that is returned from the service

Course Outline

- Chapter 1 Building RESTful Services
- Chapter 2 Designing RESTful Services
- Chapter 3 Testing RESTful Services
- Chapter 4 Securing RESTful Web Services
- Chapter 5 Cloud Design Patterns
- Chapter 6 Node.js
- Chapter 7 Node.js and Express
- Chapter 8 Promises and Testing Node with Jasmine
- Chapter 9 The FidZulu Mini Project

Course Outline

- Chapter 10 Apigee Edge
- Chapter 11 Service Virtualization
- Chapter 12 Testing with Cucumber.js
- Chapter 13 Server-Side JavaScript Programming
- Chapter 14 Functional and Reactive Programming in JavaScript
- Appendix A Building RESTful Services with JAX-RS

Course Objectives

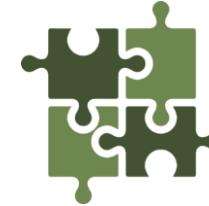
In this course, we will:

- Solve common programming problems by using design patterns
- Design and build RESTful web services
- Use Spring Boot to create RESTful web services written in Java
- Use Node.js to execute RESTful services written in JavaScript
- Use some advanced JavaScript programming techniques

Key Deliverables



Course Notes



Project Work



There is a Skills Assessment for this course

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 1: Building RESTful Services

Chapter Overview

In this chapter, we will explore:

- Understanding what a RESTful web service is
- Building RESTful services with Spring Boot
- Returning HTTP status codes

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

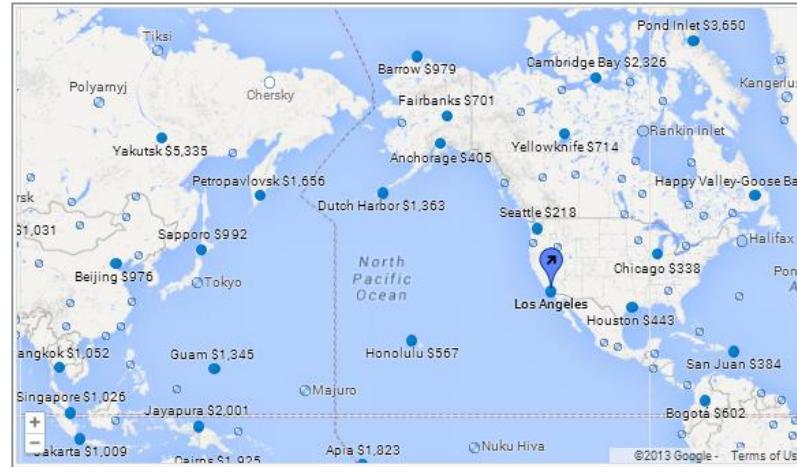
Web Application vs. Web Service

- Web applications are meant for use by human users
 - Visual representation of information, interactive
 - Multiple steps (often stateful)

The image shows a travel booking interface with tabs for Flight, Hotel, Car, and Vacation. The Flight tab is selected. It displays a search result for a round trip from Helsinki, Finland to Johannesburg, South Africa. The total price is \$1,264. The first result shows a departure from Helsinki at 1:00 p.m. on Sunday, Nov. 16, 2014, arriving in Munich at 2:35 p.m. on the same day. A 'Select' button is available for this flight. Below it, another flight option is shown departing Munich at 8:30 p.m. on Sunday, Nov. 16, 2014, arriving in Johannesburg at 8:15 a.m. on Monday, Nov. 17, 2014. There is also a 'Change Planes' link. The second result shows a departure from Helsinki at 6:30 a.m. on Sunday, Nov. 16, 2014, arriving in Munich at 8:05 a.m. on the same day.

Web Application vs. Web Service (continued)

- Web services are meant for use by automated applications
 - Machine-parseable representation of information
 - Often stateless
- Web services are “behind the scenes” and are not readily apparent
 - How do you think this “mashup” at <https://www.google.com/flights> is created?



SOAP Web Services

- A SOAP-based web service
 - Advertises a WSDL interface
 - Contains everything needed to communicate with that service
 - All the request and response messages are in XML
 - Interoperable
 - Many different clients can communicate with the service
 - The data is wrapped inside of an XML SOAP message
 - Requests
 - Responses
 - Not the most convenient format for many clients
 - Ajax
 - JavaScript

A Simpler Web Service

■ REST-based web services

- Communicate with a simpler format than SOAP-based web services
 - Simple XML
 - JavaScript Object Notation (JSON)
- No Web Service Definition Language (WSDL) interface
 - Can provide a Web Application Definition Language (WADL) interface
- Parameters often passed as part of the URL

REST-Based Web Services

Problem

- Implementing a Service-Oriented Architecture requires:
 - Interoperable, loosely coupled ways of invoking services
 - Enterprise systems consist of many applications
 - Developed using many technologies
 - How to support communication between disparate systems?

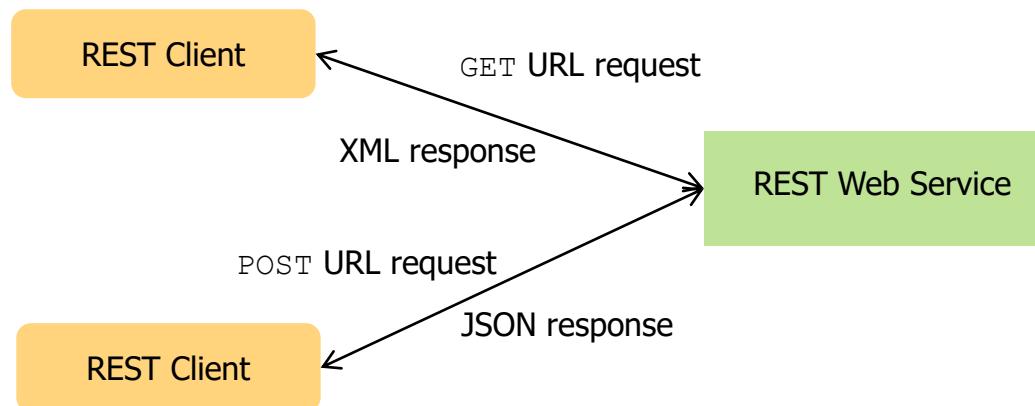
Solution

- Use HTTP as base protocol
 - Mature, standard protocol supported by all languages and operating systems
 - HTTP traffic is allowed through most firewalls
- An SOA can be implemented with REST-based web services
 - Use HTTP as the network protocol
 - Use HTTP verbs to specify operations on resources
 - GET, POST, PUT, DELETE
 - Exchange simple text-based messages
 - In JSON, XML, etc.

REST-Based Web Services (continued)

■ REST communication

- Based on HTTP requests
- Response can be in various formats
 - XML
 - JSON





HANDS-ON
EXERCISE

30 min

Exercise 1.1: Exploring the Time Service

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Spring Boot

- Provides “an opinionated view”¹ of how to build a Spring application
- Minimizes Spring configuration for many applications
- Makes it easy to create a stand-alone Spring-based application
 - That “just runs”

1. <https://spring.io/projects/spring-boot>

Spring Boot Features

- Simplifies the creation of stand-alone Spring applications
 - Provides “opinionated” starter configuration options
- Can embed a web server such as Tomcat
 - No need to deploy a war file to an external web server
- Provides production features
 - Health checks
 - Application metrics
- No code generation required
- No XML configuration required

How Does Spring Boot Work?

- It is opinionated
 - Makes reasonable default configuration settings
 - Example: a Spring Boot web application embeds the Tomcat web server
- It is customizable
 - You can modify the Maven pom file
 - Override the “reasonable default” setting with your own setting value

Starters

- Spring Boot Starter
 - A set of dependencies for a particular type of application
- Some popular starters
 - spring-boot-starter-web
 - Used to build RESTful web services
 - Uses Spring MVC and embedded Tomcat
 - spring-boot-starter-jersey
 - Used to build RESTful web services
 - Uses Apache Jersey (JAX-RS) instead of Spring MVC
 - spring-boot-starter-jdbc
 - Used for JDBC connection pooling
 - Uses Tomcat's JDBC connection pool implementation
- There are many more starters available:
 - <https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters>

Spring Initializr

- An even better approach to creating a Spring Boot project
 - <https://start.spring.io/>
- The website lets you choose options to “bootstrap your application”
 - Creates a Maven (or Gradle) Spring Boot project
 - Includes a Maven `pom.xml`
- Allows you to choose project dependencies
 - Developer tools
 - Web
 - Security
 - SQL/NO SQL
 - Testing
 - Cloud

AutoConfiguration

- Spring Boot can automatically configure your application
 - Based on the jar files in the application's classpath
 - And how the Spring managed beans are defined
- Spring Boot will examine the jar files in the classpath
 - Forms an opinion on how to configure some behavior
 - Example: if the H2 database jar file is in the classpath, and no other DataSource beans are defined, the application will be configured with an in-memory database
- Spring Boot will examine the Spring managed bean definitions
 - Example: if a JPA bean is annotated with `@Entity`, the application will be configured to use JPA without the need for a `persistence.xml` file

JPA: Java Persistence API

The All-in-One Jar File

- Spring Boot aims to create an application that “just runs”
- The application is packaged into a single, executable jar file
 - The “uber” jar file
 - All of the application dependencies are included in this jar file
- The application is launched with a command like the following:
 - `java -jar PATH_TO_EXECUTABLE/HelloWorld.jar`

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Configuration

- Here is where the Spring Boot starters help us get jump started
- A typical RESTful web service project has many dependencies
 - Spring MVC or JAX-RS
 - Tomcat
 - Jackson
 - Etc.
- We can use the `spring-boot-starter-web` to simplify this

Maven Dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The RestController

- The RESTful service controller is a Java class
 - Annotated with Spring's `@RestController`
 - Parameter to `@RequestMapping` is the first component of a URL path
- Web service methods are annotated with Spring annotations
 - `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping`
 - Parameters to annotations define remainder of URL that triggers each method
- Example: return all Widgets for request GET `http://.../warehouse/widgets`

```
@RestController
@RequestMapping("/warehouse")
public class WarehouseService {
    @GetMapping("/widgets")
    public List<Widget> queryAllWidgets() {
        ...
    }
}
```

First part of request URL

Last part of request URL

The RESTful Service GET Request

- By default, Spring automatically converts return values to JSON
 - Throwing a Spring ServerErrorException sets HTTP response status to 500

```
@GetMapping("/widgets")
public List<Widget> queryAllWidgets() {
    List<Widget> widgets;
    try {
        widgets = dao.getAllWidgets();
    }
    catch (Exception ex) {
        throw new ServerErrorException("Database error", ex);
    }
    if (widgets == null) {
        throw new ServerErrorException("No widgets in database",
                                      (Throwable) null);
    }
    return widgets;
}
```

[{"description": "Low Impact Widget", "id": 1, "price": 12.99, "gears": 2, "sprockets": 3}, ...]

Sets default HTTP status 200

Default content type of response is application/json

Spring converts Java objects to JSON

Sets HTTP status 500

The RESTful Service POST Request

- For POST and PUT requests, add a JavaBean parameter with `@RequestBody`
 - Default response content type is `application/json`
 - Wrap primitive return values in a JavaBean DTO to ensure response contains valid JSON

```
@PostMapping("/widgets")
public DatabaseRequestResult insertWidget(@RequestBody Widget widget) {
    int count = 0;
    try {
        count = dao.insertWidget(widget);
    }
    catch (Exception ex) {
        throw new ServerErrorException(
            "Error communicating with the warehouse database", ex);
    }
    if (count == 0) {
        throw new ServerWebInputException("can't insert widget " + widget);
    }
    return new DatabaseRequestResult(count);
}
```

Data Transfer Object (DTO) class

```
public class DatabaseRequestResult {
    public DatabaseRequestResult(int rowCount) { ... }
    public int getRowCount() { ... }
}
```

Sets HTTP status 400

Spring converts JavaBean DTO to valid JSON

```
{ "rowCount": 1 }
```

The Data Model

- The Data Model is a POJO
 - Not a Spring managed bean
- An instance of this class will be returned by the service
 - Converted to JSON format

The Data Model (continued)

```
public class Product {  
    private String description;  
    private Integer id;  
    private BigDecimal price;  
  
    // constructors  
  
    // getters & setters  
}
```

```
public class Widget extends Product {  
    private int gears;  
    private int sprockets;  
  
    // constructors  
  
    // getters & setters  
}
```

The Application

- The application could be packaged as a Web Application Archive (WAR) file
 - And deployed to a web server
- A simpler approach uses a stand-alone application
- Everything is packaged into an executable jar file
 - All the Java source
 - All the resource files
 - All the libraries the application depends on
- The application class uses the good old `main()` method
 - Calls the `run()` method of the `SpringApplication` class

The Application (continued)

- Annotate your application class with `@SpringBootApplication`
 - Enables Spring Boot autoconfiguration
- `scanBasePackages` tells Spring Boot which Java packages contain Spring beans
 - Can be omitted if application class is in a parent package of all other Spring beans

```
@SpringBootApplication(scanBasePackages={"com.fidelity.productservice"})
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Can be a comma-separated list

The application.properties File

- At application startup, Spring Boot reads configuration from application.properties
 - application.properties is Spring Boot's main configuration file
 - You can override default settings for Spring, Tomcat, MyBatis, SLF4J, etc.

```
server.port = 8082  
spring.database.url=jdbc:oracle:thin:@roifmrwinvm:1521/XE  
spring.database.username=scott  
spring.database.password=TIGER  
spring.database.driver-class-name=oracle.jdbc.OracleDriver  
  
mybatis.mapper-locations = classpath:com/fidelity/integration/mapper/*.xml  
mybatis.type-aliases-package = com.fidelity.business  
  
logging.level.root=warn
```

Tomcat port

Database connection settings

MyBatis configuration

Logging settings

Running the Application

- You can run the application from the command line:

```
java -jar target/warehouse-rest-service-0.1.0.jar
```

- Or you can run it using Maven:

```
mvnw spring-boot:run
```

- Logging output will be displayed
- The service should start quickly

Communicating with the Service

- Once the application has started:
 - The service will be listening for requests
- Use Insomnia to send a GET request to the web service:

The port can be set in
application.properties:
server.port = 8080

```
GET http://localhost:8080/warehouse/widgets
```

- The body of the response will look like this:

```
[ {"description": "Low Impact Widget", "id": 1, "price": 13, "gears": 2, "sprockets": 3}, ... ]
```

- Send a request with a path parameter:

```
GET http://localhost:8080/warehouse/widgets/1
```

- Response body:

```
{"description": "Low Impact Widget", "id": 1, "price": 13, "gears": 2, "sprockets": 3}
```

Handling Path Parameters

- Can inject variables in the URL path into a method parameter:

```
@GetMapping("/widgets/{id}")
public Widget queryForWidget(@PathVariable int id) {
    ...
}
```

- Example of path handled by the above method:

```
http://localhost:8080/warehouse/widgets/42
```

Query Parameters

- Query parameters in the URL can be injected into method parameters:

```
@GetMapping("/widgets")
public List<Widget> queryAllWidgets(@RequestParam String sortBy) {
    ...
}
```

- Example of URL handled by the above method:

<http://localhost:8080/warehouse/widgets?sortBy=price>

- By default, query parameters are required
 - To make a parameter optional, add `required=false`

```
@GetMapping("/widgets")
public List<Widget> queryAllWidgets(
    @RequestParam(required=false, defaultValue="price") String sortCol) {
```

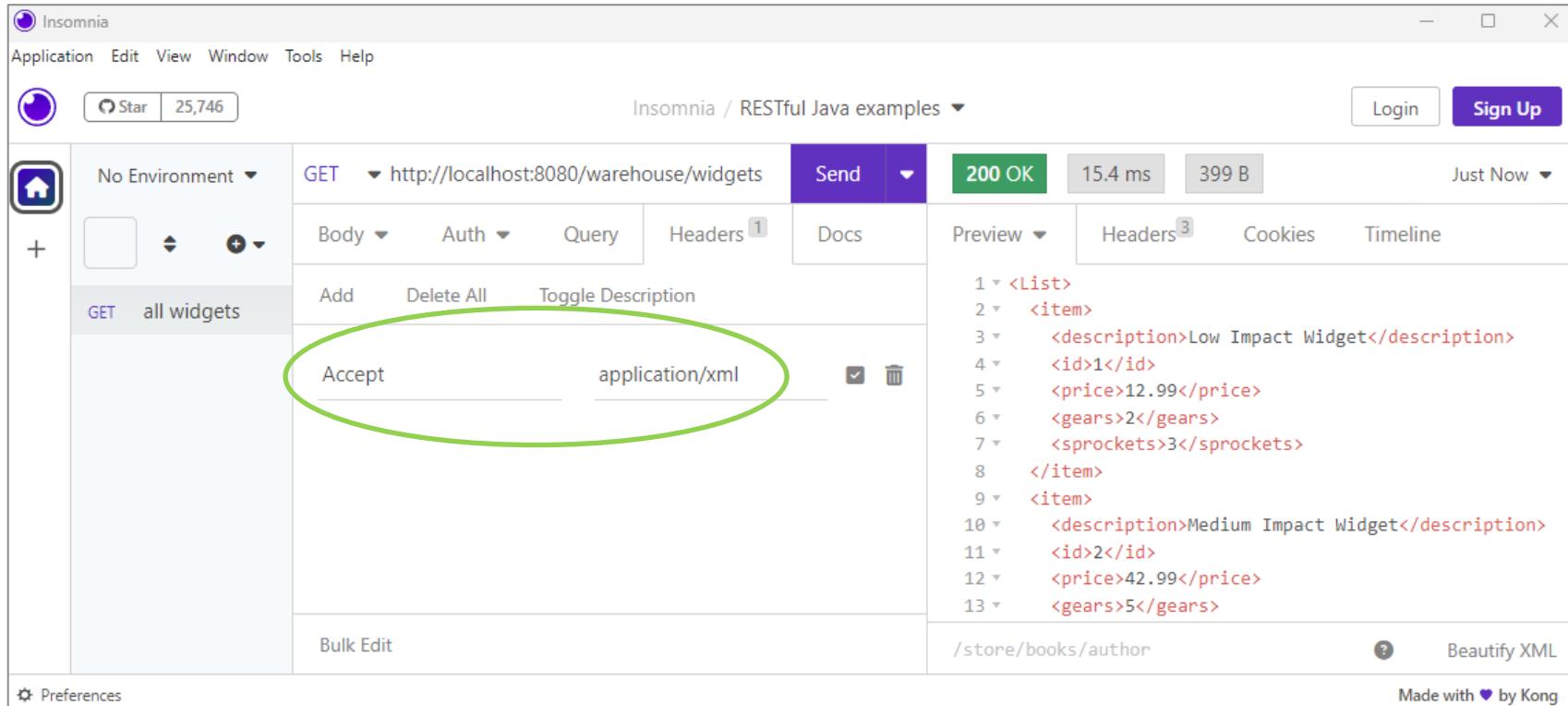
Allowed Parameter Types

- Path and Query parameters can be used on:
 - String
 - All simple types (`int`, `long`, `LocalDate`, ...)
- Other types can be handled using a custom type converter

XML and JSON

- JSON is the standard format used with REST services
- Spring handles both JSON and XML data
 - Will marshal data to/from Java classes based on type of data
- Client tells service the request data format by sending HTTP Content-Type header
 - Client says “This is the type of data *I* am sending to *you*”
 - Content-type: application/json
 - Content-type: application/xml
- Client tells service desired response data format by sending HTTP Accept header
 - Client says “This is the type of data I want *you* to send to *me*”
 - Service will automatically send data in the correct format
 - Accept: application/json
 - Accept: application/xml

Setting the Accept Header in Insomnia



The screenshot shows the Insomnia REST client interface. A green oval highlights the 'Accept' header entry in the Headers section of the request configuration.

Request Configuration:

- Method: GET
- URL: <http://localhost:8080/warehouse/widgets>
- Headers:
 - Accept: application/xml

Response Summary:

- Status: 200 OK
- Time: 15.4 ms
- Size: 399 B
- Timestamp: Just Now

Preview:

```
<List>
<item>
  <description>Low Impact Widget</description>
  <id>1</id>
  <price>12.99</price>
  <gears>2</gears>
  <sprockets>3</sprockets>
</item>
<item>
  <description>Medium Impact Widget</description>
  <id>2</id>
  <price>42.99</price>
  <gears>5</gears>
</item>
```

Navigation:

- Preview, Headers (3), Cookies, Timeline

Bottom Navigation:

- Bulk Edit
- /store/books/author
- Beautify XML

Made with ❤ by Kong

Producing JSON or XML Example

- The following method will produce XML or JSON based on `Accept` header value
- JavaBeans are converted to XML using the Jakarta XML Binding (JAXB) API
 - If you use Spring's default JAXB implementation (the Jackson library), JavaBeans don't require any modification
 - If you use a different JAXB implementation, you may need to add `@XmlRootElement` to JavaBeans

```
@GetMapping(value="/widgets/{id}",  
            produces={ MediaType.APPLICATION_JSON_VALUE,  
                      MediaType.APPLICATION_XML_VALUE })  
public Widget queryForWidget(@PathVariable int id) {  
    ...  
}
```

Not required for
Jackson library

```
@XmlRootElement  
public class Widget {  
    ...  
}
```

- Converting JavaBeans to JSON doesn't require any configuration: it just works

Producing JSON or XML Example (continued)

- The following method will accept JSON or XML based on Content-Type header:

```
@PostMapping(value="/widget",
    consumes={ MediaType.APPLICATION_JSON_VALUE,
               MediaType.APPLICATION_XML_VALUE }),
    produces={ MediaType.APPLICATION_JSON_VALUE,
               MediaType.APPLICATION_XML_VALUE })
public Widget addWidget (@RequestBody Widget widget) {
    dao.insertWidget(widget);
    return widget;
}
```

Widget parameter will be created from JSON
or XML data in the body of the message



HANDS-ON
EXERCISE

30 min

Exercise 1.2: Creating a RESTful API

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

HTTP Status Codes

- The HTTP protocol defines meaningful status codes
 - Which can be returned from a RESTful service
- Using status codes can help service consumers
 - Determine how to understand the service response
 - Especially when errors occur
- What is status code 418?

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP Status Codes (continued)

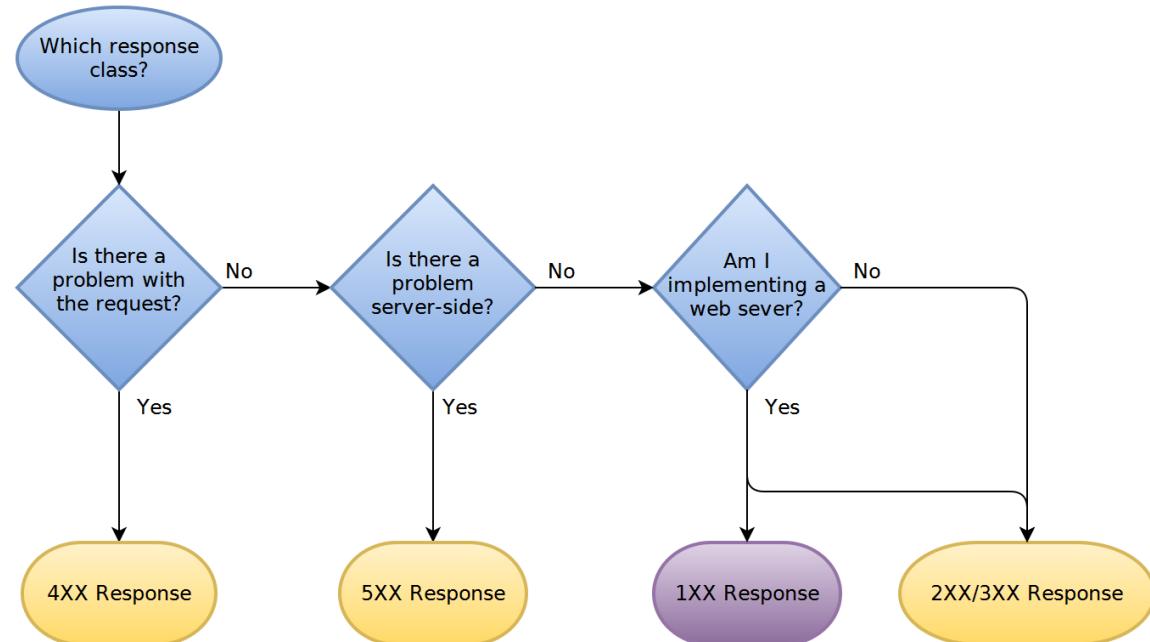
- 200 OK Response to a successful request
- 201 Created Response to POST that results in a resource creation
- 204 No Content Response to a successful request that does not return a body
- 400 Bad Request The request was malformed
- 401 Unauthorized Either invalid or missing authentication details in request
- 403 Forbidden User does not have access to the requested resource
- 404 Not Found We've all been here before
- 405 Method Not Allowed The HTTP method is not allowed for this user
- 410 Gone The resource is no longer available
- 418 ?

Why Should I Use a Status Code?

- They communicate to the RESTful client
 - When an exceptional event occurs
 - When some special behavior is required
- Many status codes represent situations that are worth handling with a special response
- Many widely used APIs are using them
 - A convention is being created
 - Following that convention makes it easier for users of your RESTful service
 - <https://gist.github.com/vkostyukov/32c84c0c01789425c29a>

What Status Should I Return?

- The following flowcharts answer this question
 - From <https://www.codetinkerer.com/2015/12/04/choosing-an-http-status-code.html>
- The flowcharts for each category of response are too big to fit on these slides
- Visit the above URL to see them





HANDS-ON
EXERCISE

Exercise 1.3: Which Status Code to Use?

20 min

- Please refer to your Exercise Manual to complete this exercise

How to Return a Status Code

■ Two main methods for returning an HTTP status code response

1. Return a `ResponseEntity` object that wraps the Java object being returned
 - Call `ResponseEntity.status()` to set the HTTP response status code
2. Throw a `ResponseStatusException` or one of its subclasses
 - All subclasses set the HTTP status code and provide an error message
 - `ServerErrorException` (code 500), `ServerWebInputException` (code 400)

■ `ResponseEntity` uses the *Builder* design pattern

- Builder pattern allows the construction of complex objects step by step
- Avoids constructors with long parameter lists
- Can provide default values for an object's properties

Example of Builder Design Pattern

- Instead of a constructor with many parameters:
- Use the Builder pattern:

```
class UserBuilder {  
    private String name = "No name";  
    private String email = "No email";  
    private String address = "No address";  
  
    public UserBuilder setName(String name) {  
        this.name = name;  
        return this;  
    }  
    public UserBuilder setEmail(String email) { ... }  
    public UserBuilder setAddress(String address) { ... }  
  
    public User build() {  
        return new User(name, age, address);  
    }  
}
```

```
class User {  
    public User(String name, String email, String addr) { ... }  
    ...  
}  
...  
User lin = new User("Lin", "1 Oak St" , "lin@wxyz.me");
```

Properties have default values

Returns this so calls can be chained

build() creates a User object

Oops! Arguments in wrong order.

```
UserBuilder builder = new UserBuilder();  
User user = builder.setName("Lin")  
    .setAddress("1 Oak St")  
    .setEmail("lin@wxyz.me")  
    .build();
```

Order of setter calls doesn't matter

Call build() instead of User constructor

Using the ResponseEntity Object

- RESTful methods can return a ResponseEntity instead of a JavaBean
 - Allows you to set response's HTTP status and headers

```
@GetMapping("/widgets")
public ResponseEntity<List<Widget>> queryAllWidgets() {
    List<Widget> widgets = dao.getAllWidgets();

    ResponseEntity<List<Widget>> response;
    if (widgets != null) {
        response = ResponseEntity.status(HttpStatus.OK)
            .body(widgets);
    }
    else {
        response = ResponseEntity.status(HttpStatus.NO_CONTENT)
            .build();
    }
    return response;
}
```

Method returns ResponseEntity

Sets status and returns a BodyBuilder

Builds response body

Builds response with an empty body

Using the ResponseEntity Object (continued)

- `ResponseEntity` defines convenience methods for common use cases
 - `ResponseEntity.ok(response-body)` – sets response status 200
 - `ResponseEntity.noContent()` – sets status 204
 - `ResponseEntity.notFound()` – sets status 404

```
@GetMapping("/widgets")
public ResponseEntity<List<Widget>> queryAllWidgets() {
    List<Widget> widgets = dao.getAllWidgets();
    ResponseEntity<List<Widget>> response;

    if (widgets != null) {
        response = ResponseEntity.ok(widgets);
    }
    else {
        response = ResponseEntity.noContent()
            .build();
    }
    return response;
}
```



HANDS-ON
EXERCISE

30 min

Exercise 1.4: Returning a Status Code

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Understanding what a RESTful web service is
- Building RESTful services with Spring Boot
- Returning HTTP status codes

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 2: Designing RESTful Services

Chapter Overview

In this chapter, we will explore:

- Designing an effective RESTful API
 - RESTful API guidelines
- Deploying a Spring Boot-based RESTful web service
- The Twelve-Factor App
- How Spring Boot is used at Fidelity
- How to debug a RESTful web service

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Using Spring Boot at Fidelity

- Fidelity developers use Spring Boot to build RESTful web services
- Documentation for using Spring Boot is provided¹
- Sample applications are available

1. <https://itec-confluence.fmr.com/display/AP119867/Springboot+Reference+Application>

Prerequisites

1. The application must be compatible with JDK v1.8
2. Spring Boot version 1.5.4.RELEASE recommended
3. Spring version 4.3.9 recommended
4. The application complies with the Twelve-Factors App guidelines
5. Recommended build tool is Maven
6. Recommended IDE is Spring Tool Suite v3.9.0.RELEASE
7. Configuration credentials must be set up in Concourse Vault for CI/CD

Note: These versions may be outdated. Check with your business unit or tech lead for current versions.

Spring Boot Reference Applications

- There are three Spring Boot reference applications
 1. RESTful service that consumes another RESTful service
 2. RESTful service protected by OAuth2 which consumes another RESTful service
 3. RESTful service with database connectivity



Exercise 2.1: Research Spring Boot at Fidelity

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

RESTful API Guidelines

- RESTful services should be stateless
- Use RESTful URLs and actions
- Prefer returning JSON instead of XML
- Support versioning
- Use token-based authentication
- Include response headers that support caching
- Use HTTP Status codes effectively
- Consider using query parameters for filtering

These are discussed
on the following slides

RESTful Services Should Be Stateless

- A RESTful service API should be stateless
- Requests should not depend on cookies
 - Or sessions
- Services are much simpler
- Services are more efficient

Designing RESTful URLs

- RESTful principles were introduced in Roy Fielding's dissertation¹
- Separate your API into a conceptual hierarchy of resources
 - Describe as nouns (not verbs)
- Use HTTP verbs to manipulate resources

GET /employees/42/email	Retrieve the email for employee 42
POST /employees/42	Create employee 42
PUT /employees/42	Replace the existing employee 42
PATCH /employees/42	Perform partial update of employee 42
DELETE /employees/42	Delete employee 42

- Fidelity requirement: for sensitive data, use PUT instead of GET

1. Architectural Styles and the Design of Network-based Software Architectures, Roy Fielding
<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

RESTful URL Guidelines

■ Design RESTful URLs that are:

1. Hierarchical
2. Entity-centric (not operation-centric)

Do this:

```
GET /warehouse/widgets
```

```
GET /warehouse/widgets/123
```

```
GET /warehouse/widgets/123/price
```

```
GET /warehouse/widgets/descriptions
```

Not this:

```
GET /warehouse/getAllWidgets
```

```
GET /warehouse/widgets/list
```

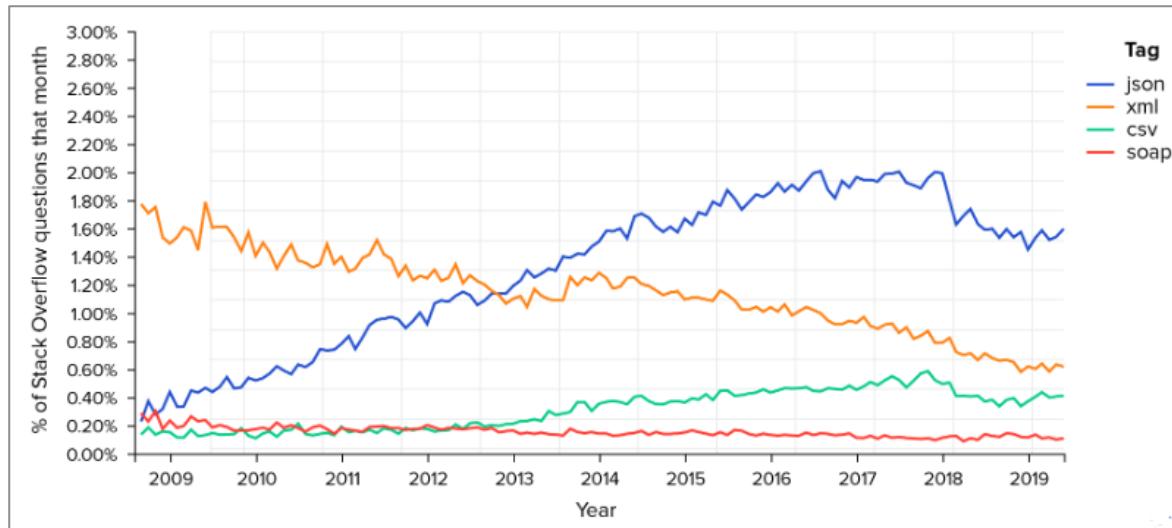
```
GET /warehouse/widgets?operation=read&id=123
```

```
GET /warehouse/read/widget/price/123
```

```
GET /warehouse/widgetDescriptions
```

Prefer JSON

- Problems with XML
 - Verbose, slower to parse, slower to convert to JavaScript objects
- JSON is simple, lightweight, easy to use in JavaScript



<https://www.toptal.com/web/json-vs-xml-part-1>

Support Versioning

- Change is inevitable
 - So, we have to deal with it
- Plan ahead for versioning support
 - Will save much time and effort later
- Where to include the version information
 - HTTP header: **API-Version: 1.1.3**
 - Request URL: GET /warehouse/**v1.1.3**/widgets/123
- Can provide support for existing clients using older versions
 - While offering new functionality for clients that need it

Token-Based Authentication

- Request authentication should be stateless
 - No dependence on sessions or cookies
- Each request should contain its authentication credentials
- More on this in the Security section

Support Caching

- HTTP provides built-in caching
- Use additional outbound response headers
 - ETag
 - Last-Modified

ETag

- The ETag HTTP header
 - Unique id for a specific version of a resource
 - Set by service in first response message
 - Contains a hash or checksum of the requested resource
 - Service will recompute ETag value whenever the resource changes
- In later requests, client sets If-None-Match header with previous ETag value
 - “If the resource’s current hash doesn’t match this hash, give me the new resource”
- Service compares ETag value in request with the resource’s current ETag value
 - If they match, service returns 304 (Not Modified) status instead of the requested resource
 - Client uses its copy of the resource instead of downloading duplicate data

Last-Modified

- Last-Modified works similarly to ETag
- Returns a timestamp as the value of the Last-Modified response header
- This is validated against the If-Modified-Since request header

Use HTTP Status Codes

- Use those status codes
 - As discussed in the Status Codes section

Filtering Resources

- Filtering, sorting, and searching can be implemented with query parameters
 - Instead of many different URLs
- Consider using a unique query parameter
 - For each field that implements filtering
- This could also be done for sorting and searching

```
GET /tickets?state=open
```

Retrieves only the tickets in the open state

```
GET /tickets?sort=priority&asc=false
```

Retrieves a list of tickets in descending order of priority

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

MyBatis Mapper Configuration for Spring Boot

- In the next exercise, MyBatis may need help finding your mapper interfaces
 - Not required previously because of `SqlSessionFactory` config in Spring config file
- Solution: add MyBatis annotations in source code
 - Add `@Mapper` to MyBatis mapper interface, or
 - Add `@MapperScan(basePackages="...")` to Spring Boot application class
- Don't store mapper interfaces in same directory as your DAO interface
 - MyBatis might mistake the DAO interface for a mapper interface
- Most MyBatis configuration is in Spring Boot's `application.properties`

```
mybatis.mapper-locations = classpath:com/fidelity/integration/mapper/*.xml  
mybatis.type-aliases-package = com.fidelity.business
```

MyBatis configuration



40 min

Exercise 2.2: Designing a RESTful Service API

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Cloud Computing As a Service

- There many categories of cloud services available for deploying your application
 - Choose based on how much control of the infrastructure you need
 - See <https://www.paulkerrison.co.uk/random/pizza-as-a-service-2-0>

Deployment Option	Description	Examples
Infrastructure as a Service (IaaS)	<ul style="list-style-type: none">• Vendor supplies hardware• You supply VMs	<ul style="list-style-type: none">• Amazon Elastic Compute Cloud (Amazon EC2)• MS Azure IaaS
Containers as a Service (CaaS)	<ul style="list-style-type: none">• Vendor supplies VM with container framework• You supply containerized apps	<ul style="list-style-type: none">• Amazon Elastic Container Service (Amazon ECS)• MS Container Service
Platform as a Service (PaaS)	<ul style="list-style-type: none">• Vendor supplies VMs and app servers• You deploy web apps	<ul style="list-style-type: none">• Google Cloud• MS Azure PaaS
Function as a Service (FaaS)	<ul style="list-style-type: none">• Vendor supplies runtime environment• You write functions	<ul style="list-style-type: none">• Amazon Lambda• Google Cloud Functions
Software as a Service (SaaS)	<ul style="list-style-type: none">• Vendor supplies software• You supply data	<ul style="list-style-type: none">• MS O365• Adobe Creative Cloud

Deploying to Cloud Foundry

- Cloud Foundry – open-source cloud application framework
 - Allows you to support PaaS/CaaS on your own hardware
 - Creates and manages containers
 - Used within Fidelity to create private clouds behind the company's firewall
- Cloud Foundry employs a “buildpack” approach
 - Similar to the Spring Boot “uber” jar file
 - The buildpack wraps deployed code in what is needed to start the application
 - The Java buildpack provides support for Spring Boot applications
- Use the Cloud Foundry `cf` command line tool
 - Deploy the Spring Boot application by using the `cf push` command
 - This will upload the uber jar file to Cloud Foundry
- Cloud Foundry uploads and deploys the application
 - Once that process completes successfully, the application is live!

Deploying to Amazon Web Services (AWS)

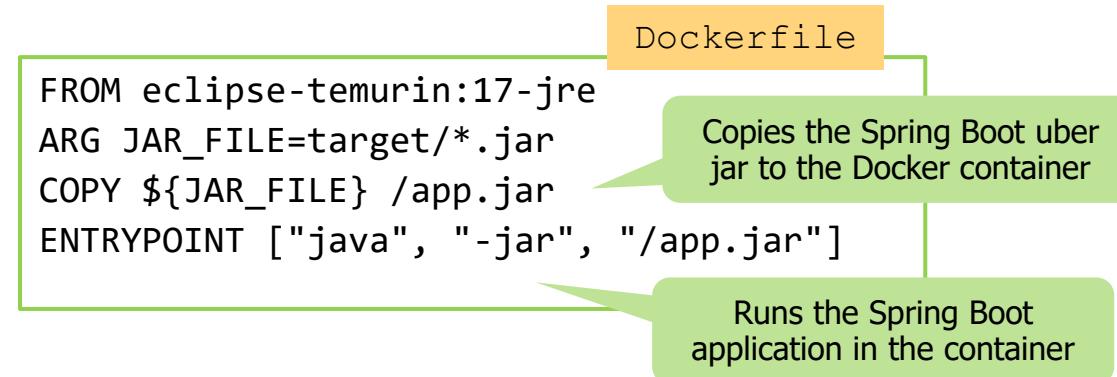
- Amazon Web Services provides multiple ways to install a Spring Boot application
 - Either as a war file or as an uber jar file
- AWS Elastic Beanstalk is the simplest option
 - Load balanced by default
 - Supports two options for a Java application
 - Tomcat platform
 - Java SE platform
- Tomcat platform
 - Supports web applications deployed as a war file
- Java SE platform
 - Supports Spring Boot uber jar file applications
 - Runs an nginx instance on port 80 to proxy the application
 - The application runs on port 5000

Deploying with Docker

- Docker – open-source container framework
 - Container: lightweight VM with only the libraries essential to run your application
 - Containers start and stop quickly, make it easy to scale an application
- Define the structure of your container using a simple Dockerfile file format
 - Specifies the “layers” of a Docker image
- A Docker image consists of read-only layers
 - Each layer represents an instruction
 - Each layer is a delta of the changes from the previous layer
- Pre-built images can be pulled from the public Docker Hub repository
 - <https://hub.docker.com/>
- Best practices for creating a Docker file are available online
 - https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

A Simple Docker File

- FROM – sets the base Docker image
 - Eclipse Temurin is a certified binary build of OpenJDK
- ARG – defines a variable
- COPY – copies files from the local file system to the container's file system
- ENTRYPOINT – an executable to be run when the container starts



Building a Docker Image

- A Docker image can then be built using either Maven or Docker:

```
mvn install dockerfile:build
```

```
docker build --tag image-name:image-tag .
```

Name for the image in
a container repository

Optional tag; e.g.,
version number

- For Maven builds, pom.xml needs the following:

- The dockerfile-maven-plugin added to the <plugins> section
- The maven-dependency-plugin is configured to unpack the uber jar

Running a Docker Image

- The Docker image can then be run with the following command:

```
docker run -p 8081:8080 image-name
```

Port number
on
local machine

Port number
in
container

- The Spring Boot application is now available on `http://localhost:8081`
 - Docker maps requests to port 8081 to the container's port 8080
 - A CaaS framework scales the app by starting additional containers on different ports

Optional Exercise 2.3: Deploying a Spring Boot RESTful Service into Docker



20 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

The Twelve-Factor App¹

- A methodology for building software-as-a-service applications
- Uses declarative formats for setup automation
 - Makes it easier for new developers joining the project
- Has a clean contract with the underlying operating system
 - Supports ease of portability between environments
- Suitable for deployment on modern cloud platforms
 - Minimizes the need for servers and administrators
- Minimizes divergence between development and production systems
 - Enables continuous deployment
- Can scale up easily
 - Without significant changes to tooling, architecture, or development

1. <https://12factor.net/>

The Twelve Factors

- I. Codebase
 - One codebase tracked in version control
 - Many deploys
- II. Dependencies
 - Explicitly declared and isolated
- III. Configuration
 - Store configuration information in the environment
- IV. Backing services
 - Treat backing services as attached resources
- V. Build, release, run
 - Strictly separate the build and run stages



The Twelve Factors (continued)

VI. Processes

- Execute the application as a stateless process

VII. Port binding

- Export services via port binding

VIII. Concurrency

- Scale out the application via the process model

IX. Disposability

- Fast startup
- Graceful shutdown

X. Development/Production Parity

- Keep development, staging, and production as similar as possible

The Twelve Factors (continued)

XI. Logs

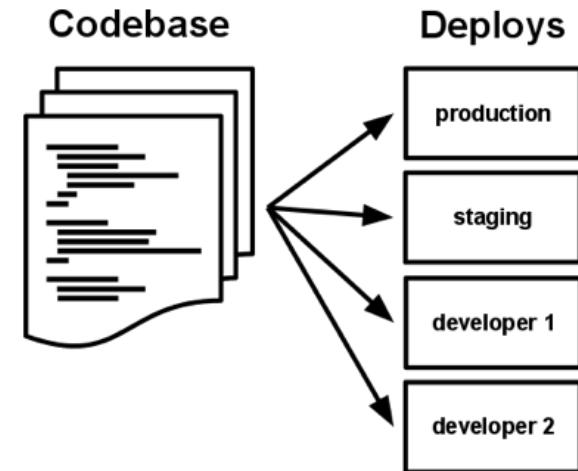
- Treat logs as event streams

XII. Administrative processes

- Run administrative/management tasks as one-off processes

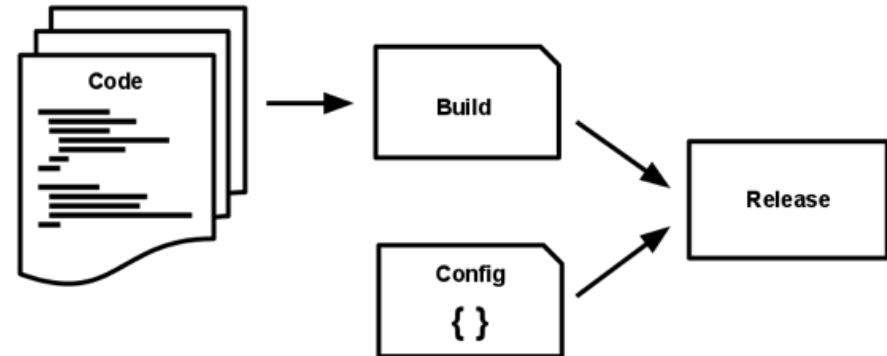
One Codebase

- One codebase, many deploys
- Only one codebase per application
- A deploy of the application is a running instance of the application
 - Production
 - Staging
 - Developers
- Codebase is tracked in a version control system



Build, Release, Run

- A codebase is transformed into a deploy through three stages
 - Build stage
 - Converts codebase into an executable
 - Release stage
 - Combines the build with the deploy's current configuration
 - Ready for immediate execution
 - Run stage
 - Runs the application in an execution environment



Processes

- The application is executed as one or more processes
- Twelve-factor processes:
 - Stateless
 - Share nothing
 - Data to be persisted are stored in a backing service (i.e., database)
- Session state data
 - No “sticky sessions”
 - Should be stored in a datastore with time expiration



30 min

Optional Exercise 2.4: Research the Twelve-Factor App

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

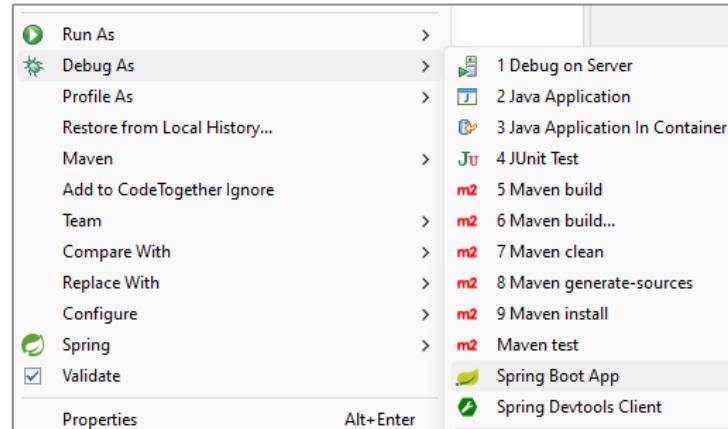
The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Debugging RESTful Services

- Debugging is one of the most important tools/skills for writing successful software
- Debugging a Spring Boot-based RESTful service in STS is easy:
 - Set breakpoints in the RESTful controller methods, business service methods, etc.
 - Right-click **project > Debug As > Spring Boot App**
 - Use Insomnia or a client UI to send an HTTP request to the service
 - Step through the service method in using the debug toolbar





30 min

Exercise 2.5: Debugging a RESTful Service

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Configuring MyBatis for Spring Boot

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Designing an effective RESTful API
 - RESTful API guidelines
- Deploying a Spring Boot-based RESTful web service
- The Twelve-Factor App
- How Spring Boot is used at Fidelity
- How to debug a RESTful web service

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 3: Testing RESTful Services

Chapter Overview

In this chapter, we will explore:

- The responsibilities of a RESTful Service Controller
- Performing unit testing of POJOs in the back end
- Testing the RESTful Service controller in the web (HTTP) environment
- Performing end-to-end testing of a RESTful Service using `TestRestTemplate`

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

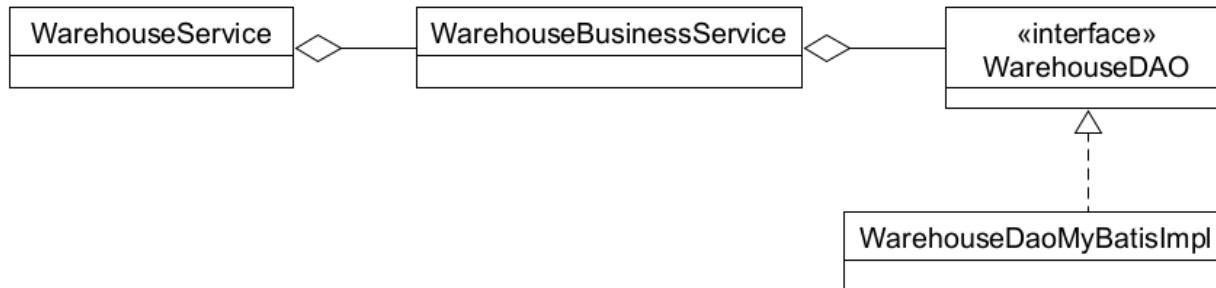
Chapter Summary

A RESTful Web Service Front End

- A RESTful web service is a front end to some existing resource
 - Provides a simple means of accessing and managing the existing resource
 - Typically, this is an HTTP interface
 - Text based, accessible to any type of client
- The RESTful service code is generally fairly simple
 - Management of the existing resource is provided by other classes
 - Often a DAO
 - Or a Business Service

RESTful Service

```
@RestController  
@RequestMapping("/warehouse")  
public class WarehouseService {  
    @Autowired private WarehouseBusinessService service;  
  
    @PostMapping("/widgets") public DatabaseRequestResult insertWidget(@RequestBody Widget widget) {  
        int count = 0;  
        try {  
            int count = service.insertWidget(widget);  
            return new DatabaseRequestResult(count);  
        } catch (Exception e) {  
            throw new ServerErrorException("Database problem", e);  
        }  
    }  
}
```



Processing a Request to a RESTful Service

#	Responsibility	Description
1	Listen for HTTP requests	The service should respond to the URLs specified in the web methods
2	Deserialize the inputs	Parse the incoming request and create Java objects from path and request parameters and the request body
3	Validate the input	Validate the incoming input in the request
4	Call the backend	Call on the backend to process the input; handle exceptions from the backend
5	Serialize the output	Serialize the output from the back end into an HTTP response
6	Translate exceptions	Translate any exceptions into a meaningful error message and HTTP status

Unit or Integration Test of a RESTful Service

- The only step that would be possible in a unit test of a RESTful controller would be Step 4, “Call the backend” using a mock
 - The other steps are provided by Spring Boot
- Problem: a simple unit test will not cover the HTTP layer at all
 - But HTTP processing is the RESTful service’s primary responsibility
- Solution: an integration test with Spring will provide the HTTP layer
 - Spring provides all the managed beans that the web service controller needs
 - Spring provides all the framework beans for all the other steps
 - Your RESTful controller behaves as if it’s processing real HTTP requests
- Spring Boot provides the `@WebMvcTest` annotation
 - This creates and uses a Spring application context that contains only the managed beans that are necessary for testing the web service controller

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Testing a RESTful Service Project

- There are several techniques used to test the components of a RESTful service project
 1. Testing the back-end functionality with unit and integration tests
 - Data access objects
 - Business services
 2. Testing the RESTful service controllers running in the web layer
 - Provide complete testing coverage of the HTTP request handling
 3. End-to-end testing
 - Perform end-to-end testing of the RESTful service

1. Testing Back End Functionality

1. True unit testing of the POJOs (not REST controllers) in the project
 - Spring is not used for configuration
 - This allows complete testing coverage of the business functionality
 - Dependencies can be mocked
2. Integration testing of POJOs
 - Verify the Spring configuration is correct
 - Verify the POJO components work together correctly

2. Testing the RESTful Service Controllers

- Spring provides support for running tests on web services with `@WebMvcTest`
 - Spring Boot instantiates only the beans necessary for the web layer
- If your service defines only one controller, just add `@WebMvcTest` to the test class
 - Spring scans the project and finds the REST controller class

```
@WebMvcTest  
class WarehouseControllerWebLayerTest {
```

- If your service has multiple controllers, specify the controller for each test class

```
@WebMvcTest(WarehouseController.class)  
class WarehouseControllerWebLayerTest {
```

3. End-to-End Testing of the RESTful Service

- Spring provides `TestRestTemplate` for testing actual HTTP calls to the web service
- This is integration testing of the RESTful API
 - Tomcat runs in a separate thread
 - The full Spring configuration is utilized

Spring Test Types

	DAO Integration Test	POJO Integration Test	POJO Unit Test	Web Layer Test	End-to-End Test
Purpose of test	Verify DAO behavior and Spring configuration	Verify backend POJO behavior and Spring configuration	Test any class in complete isolation	Verify Spring RESTful service class interaction with HTTP requests	Verify integration of all components of a Spring RESTful service implementation
Tests use Spring config?	Yes	Yes	No	Yes	Yes
Tests require DB setup?	Yes	Yes	No	No	Yes
Tests use mocks?	No	No	Yes	Yes	No
Needs external server?	No	No	No	No	No (but it can be configured to use an external server)
Runs embedded server?	No	No	No	No	Yes (unless it's configured to use an external server)
Spring features used	@SpringBootTest @Transactional	@SpringBootTest @Transactional		@WebMvcTest @MockBean MockMvc	@SpringBootTest() webEnvironment=... @Sql
Advantages	Supports thorough automated testing of database code, including Spring configuration.	Supports thorough automated testing of other backend components, including their Spring configuration.	Supports 100% test coverage of all code, including error conditions, for any class.	Verifies Spring configuration of RESTful endpoint. Supports 100% test coverage of all code, including error conditions, for a RESTful endpoint.	Tests Spring configuration of all components. Tests interactions of all components.
Disadvantages	May be difficult to get 100% coverage, especially for some error conditions.	May be difficult to get 100% coverage, especially for some error conditions.		Setup can be complex. Verifying expectations may be complex.	May be difficult to get 100% coverage, especially for some error conditions.

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

DAO Test

- We often start by testing the integration tier first
- To verify that the DAO is working, we'll need to query the database directly
- We can use Spring's `JdbcTestUtils` class
 - Defines methods `countRowsInTableWhere()` and `deleteFromTables()`

```
import static org.springframework.test.jdbc.JdbcTestUtils.countRowsInTableWhere;  
  
{@SpringBootTest  
@Transactional  
public class WarehouseDaoIntegrationTest {  
    @Autowired  
    private WarehouseDao dao;          For executing SQL queries  
    @Autowired  
    private JdbcTemplate jdbcTemplate;   Collection of all widgets in database  
    private List<Widget> expectedWidgets = List.of(new Widget(...), new Widget(...), ...);  
}}
```

Spring will inject the DAO implementation

DAO Test (continued)

- Test cases verify database contents before and after DAO method calls

```
@Test  
void testGetAllWidgets() {  
    List<Widget> widgets = dao.getAllWidgets();  
    assertEquals(allWidgets, widgets);  
}  
  
@Test  
void testDeleteWidget() {  
    int id = 1;  
    // verify that Widget 1 is in the database  
    int rowsWithId1 = countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id);  
    assertEquals(1, rowsWithId1);  
  
    int rows = dao.deleteWidget(id);  
  
    assertEquals(1, rows);  
    // verify that Widget 1 is no longer in the database  
    rowsWithId1 = countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id);  
    assertEquals(0, rowsWithId1);  
}
```

Table name

WHERE condition

Generated SQL:
SELECT count(*) from widgets where id = 1

Mock Testing

- Mocks are often used to construct true unit tests of POJOs
 - No Spring configuration will be used (that would produce an integration test)
- The first step in testing a web service project:
 - Complete unit testing coverage of the functionality provided by POJOs
 - Business services
 - Data access objects
 - Mock the dependencies of a POJO
- The implementation of a web service is primarily calling on a POJO
 - It is not necessary to mock the dependencies of the web service
 - We will gain test coverage of the web service in our later tests

POJO Test

- POJO tests make sense for business objects with dependencies on DAOs, other services, etc.
- Often a business service will have a dependency on a DAO

```
@Service  
@Transactional  
public class WarehouseBusinessService {  
    @Autowired  
    private WarehouseDao dao;  
  
    public List<Widget> findAllWidgets() {  
        List<Widget> widgets;  
        try {  
            widgets = dao.getAllWidgets();  
        }  
        catch (Exception e) {  
            String msg = "Error querying the Warehouse database."  
            throw new WarehouseDatabaseException(msg, e);  
        }  
        return widgets;  
    }  
}
```

Business service has a dependency on the DAO

The method we want to test

The method we're testing calls a DAO method

Unit Test Mocking the Dependency

```
import org.mockito.Mockito.*;  
  
public class WarehouseBusinessServiceTest {  
    @Mock WarehouseDao mockDao;  
    @InjectMocks WarehouseBusinessService service;  
  
    @BeforeEach  
    void setUp() throws Exception {  
        MockitoAnnotations.openMocks(this);  
    }  
  
    @Test  
    void testFindAllWidgets() {  
        List<Widget> expectedWidgets = List.of(  
            new Widget(1, "Low Impact Widget", 12.99, 2, 3),  
            new Widget(2, "High Impact Widget", 15.99, 4, 5));  
  
        when(mockDao.getAllWidgets())  
            .thenReturn(expectedWidgets);  
  
        List<Widget> actualWidgets = service.findAllWidgets();  
        assertEquals(expectedWidgets, actualWidgets);  
    }  
}
```

The usual Mockito configuration

Mockito calls the service constructor and injects the mock DAO

Initialize the list of Widgets that will be returned by the mock DAO

Set the return value of the mock DAO's getAllWidgets()

Service's findAllWidgets() calls DAO's getAllWidgets()

Integration Testing

- After unit tests are defined for the back end POJOs, integration tests can be written
 - Use Spring to manage and inject the dependencies
- For example, a business service has a dependency on a DAO
 - An integration test can verify the business service and DAO work together successfully

Business Service Integration Test

```
@SpringBootTest
@Transactional
class WarehouseBusinessServiceIntegrationTest {
    @Autowired
    WarehouseBusinessService service;

    // Because the test database is tiny, we can check all products.
    // If the database was larger, we could just spot-check a few products.
    private static List<Widget> allWidgets = List.of(
        new Widget(1, "Low Impact Widget", 12.99, 2, 3),
        new Widget(2, "Medium Impact Widget", 42.99, 5, 5),
        new Widget(3, "High Impact Widget", 89.99, 10, 8)
    );

    @Test
    void testGetAllWidgets() {
        List<Widget> widgets = service.findAllWidgets();

        assertEquals(allWidgets, widgets);
    }
    ...
}
```



Exercise 3.1: Testing Back-End POJOs

45 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Testing the Web Layer

- It is not enough to verify the web service returns the correct HTTP status
 - We need an integration test for Spring configuration and HTTP request handling
 - But with mock HTTP requests and mock dependencies (business service, DAO)
- Use `@WebMvcTest` to narrow testing down to just the web layer
- `@WebMvcTest` auto-configures the Spring MVC infrastructure
 - But it limits which beans it scans for
 - For example, `@WebMvcTest` scans for `@RestController` but doesn't scan for `@Component`
- The test assertions will be the same as in an end-to-end application test
- Spring's `MockMvc` object will send mock HTTP requests to the controller being tested

Testing with @WebMvcTest

```
@WebMvcTest
public class WarehouseServiceWebMockTest {
    @Autowired private MockMvc mockMvc;
    @MockBean WarehouseBusinessService mockBusinessService;

    @Test
    public void testAddWidgetToWarehouse() throws Exception {
        Widget w = new Widget(42, "Test widget", 4.52, 20, 10);
        when(mockBusinessService.addWidget(w)).thenReturn(1);
        ObjectMapper mapper = new ObjectMapper();
        String jsonString = mapper.writeValueAsString(w);

        mockMvc.perform(post("/warehouse/widgets")
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON)
                .content(jsonString))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.rowCount").value(1));
    }
}
```

MockMvc sends a
mock POST request

Populate request
header and body

Log request and
response to console

Convert JavaBean to
JSON using Jackson's
ObjectMapper

Test for status 200

Read JSON using JsonPath

Testing JSON Responses with JsonPath

- Test the contents of JSON responses using Spring's `JsonPathResultMatchers`
 - Uses JsonPath expressions to read values of a JSON string
 - JsonPath: <https://github.com/json-path/JsonPath>
- Example: read an array of widgets serialized as JSON

```
[ {"description": "Low Impact Widget", "id": 1, "price": 13, "gears": 2, "sprockets": 3},  
 {"description": "Medium Impact Widget", ...}, { ... }, ... ]
```

`$` – entire JSON string (the array of objects)
`$[0]` – first array item (a widget object)
`$[0].id` – `id` element value of first array item

- Example: get `rowCount` value of a JSON string with one object
 - `$` – entire JSON string (the object)
 - `$.rowCount` – value of `rowCount` element of the object

```
{ "rowCount": 1 }
```

Testing Exception Handling

- Write negative test cases by configuring the mock service method
 - Throw an exception
 - Return an empty list
 - Return null

```
@Test  
public void testQueryForAllWidgets_ServiceThrowsException() throws Exception {  
    when(service.findAllWidgets())  
        .thenThrow(new RuntimeException());  
  
    mockMvc.perform(get("/warehouse/widgets"))  
        .andDo(print())  
        .andExpect(status().is5xxServerError())  
        .andExpect(content().string(is(emptyOrNullString())));  
}
```

Mock business service throws an exception

Verify controller handles exception correctly

Matchers from the Hamcrest framework



Exercise 3.2: Testing the Web Layer

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

End-to-End Testing with TestRestTemplate

- We should write some tests that assert the actual behavior of the web service
 - Start the application and listen for a connection
 - Send a request to the web service
 - Assert the response
- TestRestTemplate
 - Sends real HTTP requests with request headers, a body, etc.
 - Provided by `@SpringBootTest`
 - Just autowire it into your test class
- Add `webEnvironment` parameter to `@SpringBootTest` to start a Tomcat instance
 - Spring deploys the complete application to Tomcat
 - Use `WebEnvironment.RANDOM_PORT` to avoid port conflicts
 - The randomly selected port will automatically be used in the web request URL

Testing the Service with TestRestTemplate

- Note the use of `@Sql` on the class to execute the database setup scripts
 - Because `@SpringBootTest` runs Tomcat in a different thread than the test cases themselves, `@Transactional` has no effect here
 - So, we need to re-initialize the database before each test case

```
@SpringBootTest(classes=WarehouseServiceApplication.class,  
                 webEnvironment=WebEnvironment.RANDOM_PORT)  
  
@Sql(scripts={"classpath:schema-dev.sql", "classpath:data-dev.sql"})  
  
public class WarehouseServiceTestRestTemplateTest {  
    @Autowired  
    private TestRestTemplate restTemplate;  
    @Autowired  
    private JdbcTemplate jdbcTemplate;  
    ...
```

Start Tomcat in a separate process

Execute these database scripts before each test

Testing a GET Request

```
@Test  
public void testQueryForAllWidgets() {  
    Widget widget1 = new Widget(1, "Low Impact Widget", 12.99, 2, 3);  
    Widget widget3 = new Widget(3, "High Impact Widget", 89.99, 10, 8);  
  
    // get the row count from the Widgets table  
    int rowCount = countRowsInTable(jdbcTemplate, "widgets");  
  
    String requestUrl = "/warehouse/widgets";  
  
    ResponseEntity<Widget[]> response =  
        restTemplate.getForEntity(requestUrl, Widget[].class);  
  
    assertEquals(HttpStatus.OK, response.getStatusCode());  
  
    // verify that the service returned all Widgets in the database  
    Widget[] responseWidgets = response.getBody();  
    assertEquals(rowCount, responseWidgets.length);  
  
    // spot-check a few Widgets  
    assertEquals(widget1, responseWidgets[0]);  
    assertEquals(widget3, responseWidgets[2]);  
}
```

Send a GET request

Convert JSON response to an array of Widgets

Testing a POST Request

```
@Test
public void testAddWidgetToWarehouse() {
    int rowsBeforeInsert = countRowsInTable(jdbcTemplate, "widgets");
    int id = 42;
    Widget widget = new Widget(id, "Test widget", 4.52, 20, 10);
    String requestUrl = "/warehouse/widgets";
    ResponseEntity<DatabaseRequestResult> response =
        restTemplate.postForEntity(requestUrl, widget, DatabaseRequestResult.class);

    // verify the response HTTP status and response body
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(1, response.getBody().getRowCount()); // {"rowCount": 1}

    // verify that one row was added to the Widgets table
    int rowsAfterInsert = countRowsInTable(jdbcTemplate, "widgets");
    assertEquals(rowsBeforeInsert + 1, rowsAfterInsert);

    // verify that the new widget is in the Widgets table
    assertEquals(1, countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id));
}
```

Send a POST request

Serialize this object as JSON in the request body

Convert the JSON in the response to this DTO type

Testing for Errors

```
@Test
public void testAddWidget_DuplicateKey() {
    int rowsBeforeInsert = countRowsInTable(jdbcTemplate, "widgets");
    int dupeId = 1;
    Widget widget = new Widget(dupeId, "Test widget", 4.52, 20, 10);

    String requestUrl = "/warehouse/widgets";
    ResponseEntity<DatabaseRequestResult> response =
        restTemplate.postForEntity(requestUrl, widget, DatabaseRequestResult.class);

    // verify the response HTTP status
    assertEquals(HttpStatus.INTERNAL_SERVER_ERROR, response.getStatusCode());

    // verify that no rows were added to the Widgets table
    assertEquals(rowsBeforeInsert, countRowsInTable(jdbcTemplate, "widgets"));
}
```

New widget has same id as existing widget

Testing for Exceptions

```
@Test
public void testInsertWidget_DuplicatePrimaryKey() {
    // drop the Widgets table to force a database exception
    JdbcTestUtils.dropTables(jdbcTemplate, "widgets");

    String request = "/warehouse/widgets/99";

    ResponseEntity<Widget> response = restTemplate.getForEntity(request, Widget.class);

    // verify the response HTTP status
    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.INTERNAL_SERVER_ERROR)));
}
```

Exercise 3.3: End-to-End Testing with TestRestTemplate



30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The responsibilities of a RESTful Service Controller
- Performing unit testing of POJOs in the back end
- Testing the RESTful Service controller in the web (HTTP) environment
- Performing end-to-end testing of a RESTful Service using `TestRestTemplate`

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 4: Securing RESTful Web Services

Chapter Overview

In this chapter, we will explore:

- Techniques for securing RESTful web services
- Session management practices
- Authentication and authorization
- The details of OAuth for authorization

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

Session Management

- RESTful services should use session-based authentication
 - Establish a session token via POST
 - Or use an API key as a POST body argument
- Critical information should not appear in the URL
 - Username
 - Password
 - Session token
 - API key

OWASP has a series of “cheat sheets” for developers:

<https://cheatsheetseries.owasp.org/>

For more information about RESTful security, see the following:

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Protect Session State

- RESTful services are designed to be stateless
- Use only a session token or API key
 - To maintain client state in a server-side cache
- To prevent replay exploits
 - Use a time limited key or token

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

RESTful Authentication

- Several techniques to accomplish this
 - Basic authentication
 - HMAC
 - OAuth

Basic Authentication

- The simplest way to manage authentication
 - Use HTTP basic authentication
- Username and password are passed in the HTTP header
 - But **not** encrypted
- Use only with SSL/TLS

HMAC Authentication

- Hash-based Message Authentication Code (HMAC)
 - Uses a secure *hash function* to create a “fingerprint” of a string
 - Popular hash function: SHA-256 (Secure Hash Algorithm that produces a 256-bit output)
 - It’s practically impossible to reverse-engineer the original input from the hashed output
- HMAC sends a hashed version of the password
 - With some other information
 - Such as a timestamp, a nonce, or the hash of the message body
 - To help prevent (or at least detect) any tampering of the message body
- Nonce
 - A number that is used only once
 - To prevent replay attacks

OAuth

- “An open protocol to allow secure authorization in a simple and standard method from web, mobile, and desktop applications.”¹
- Enables third-party applications to obtain limited access to a web service
 - Known as “secure designated access”
- Example scenario:
 - You can grant ESPN.com the right to access your Facebook profile
 - Without knowing your Facebook password
- OAuth does not share passwords
 - Instead, it uses authorization tokens
 - Tokens prove an identity between consumers and service providers

1. <https://oauth.net/>

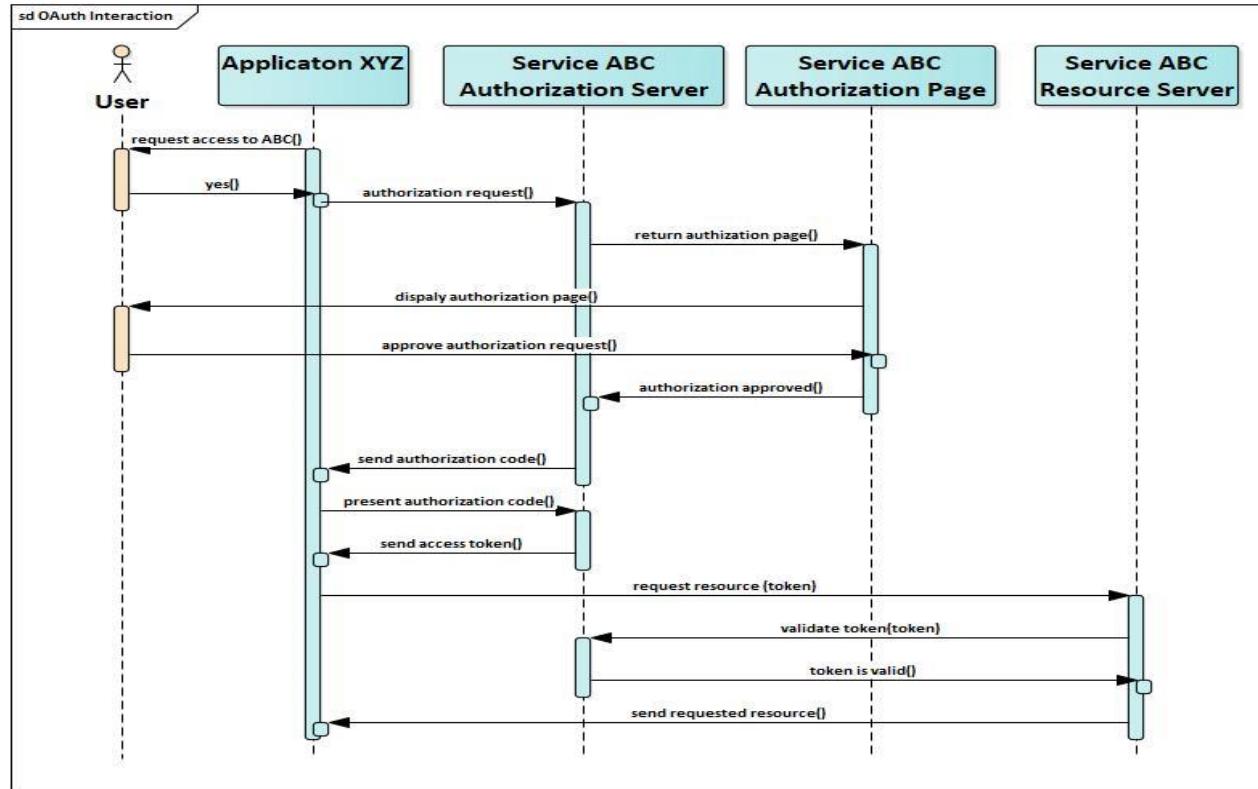
Application Registration

- Before using OAuth, the application must register with the service
 - Using the “developer (API)” portion of the service website
- The following information must be provided:
 - Application name
 - Application website
 - Redirect URI
 - The service will redirect the user to this location after authorization
 - This is the part of your application that will handle authorization codes
- Once the application is registered, the service issues client credentials
 - Client identifier
 - Publicly known string that identifies the application
 - Client secret
 - Used to authenticate the identity of the application to the service API

Authorization Grant

- There are four types of grant types supported by OAuth2
- Authorization Code
 - Used with server-side application
 - Most commonly used
- Implicit
 - Used with mobile applications or web applications that run on the user's device
- Resource Owner Password Credentials
 - Used with trusted applications
 - Like applications owned by the service
- Client Credentials
 - Used with applications API access

OAuth Authorization Code Flow



This diagram is based on RFC 6749, 4.1 <https://tools.ietf.org/html/rfc6749#section-4.1>

Stepping Through the Authorization Code Flow

1. Application XYZ asks the user for access to resources provided by Service ABC.
 - a. The user agrees.
2. Application XYZ sends an authorization request to Service ABC Authorization Server.
 - a. Using its authorization endpoint.
3. Service ABC returns an authorization page.
4. The authorization page is displayed to the user.
5. The user approves the requested permissions.
 - a. Authorization page form is submitted to the Service ABC Authorization Server.
6. Service ABC Authorization Server issues a short-lived authorization code.
 - a. Sent to application XYZ.

Stepping Through the Authorization Code Flow (continued)

7. Application XYZ presents the authorization code to Service ABC Authorization Server.
 - a. Sent to Service ABC Authorization Server token endpoint.
8. Service ABC Authorization Server issues an access token for application XYZ.
9. Application XYZ requests the desired resource from Service ABC Resource Server.
 - a. Presents the access token.
10. Service ABC Resource Server requests verification of access token.
 - a. Sends access token to Service ABC Authorization Server.
11. Service ABC Authorization Server returns confirmation of the access token.
12. Service ABC Resource Server returns the requested resource to application XYZ.

Demo and Exercise 4.1: Using OAuth2 for Authorization



30 min

- The instructor will demonstrate the usage of the OAuth2 setup
- After the demonstration, experiment with the OAuth2 related services
 - Explore the “outage” of one or more services and the succinct error messages
 - Experiment with changing parameters

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

RESTful Web Security

- The following recommendations are based on the OWASP REST Security Cheat Sheet:
 - HTTPS
 - Security tokens
 - API keys
 - Restrict HTTP methods
 - Input validation
 - Security headers
 - HTTP return codes

For more information about RESTful security, see the following:
https://www.owasp.org/index.php/REST_Security_Cheat_Sheet

HTTPS

- Secure RESTful services should provide only HTTPS endpoints
- This protects all data transferred between the client and the service
- For highly sensitive or privileged services, consider using client-side certificates

Security Tokens

- RESTful services can require a security token from the client
 - The token can be used for access control decisions
- JSON Web Tokens are often the preferred format for security tokens
 - <https://tools.ietf.org/pdf/rfc7519.pdf>
 - A cryptographic signature is the recommended way to protect the integrity of the security token
- Contents of a JSON Web Token:
 - Issuer – is this a trusted issuer?
 - Audience – is the relying party in the target audience for this token?
 - Expiration time – is the token still valid?
 - Not before time – is the token valid yet?

JSON Web Tokens (JWT)

When to use JSON Web Tokens?

- Authorization
 - When the user logs in, the JWT will be generated and returned
 - Each subsequent request includes the JWT
- Information exchange
 - Transmits information securely
 - Signing the JWT with a private key assures the receiver who the sender is
 - Also verifies the content has not been modified

The contents (header, payload) of a JWT can be decoded and read by anyone

Encrypt sensitive data in the payload

JSON Web Token Structure

- JSON Web Tokens contain three parts separated by dots (.)
 - Header
 - The type of the token (JWT)
 - The signing algorithm (HMAC, SHA256, RSA, ...)
 - Payload
 - Contains claims about an entity (usually the user)
 - Signature
 - Calculated from the encoded header, encoded payload, and a secret
 - Using a specified algorithm

Example JWT

■ Header

- { "alg": "HS256", "typ": "JWT" }

■ Payload

- { "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }

■ Signature

- HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

■ JWT

- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o



Exercise 4.2: JSON Web Tokens (JWT)

20 min

- Please refer to your Exercise Manual to complete this exercise

API Keys

- API keys can reduce the impact of a denial-of-service attack by doing the following:
 - Require an API key for every request to the service
 - Return a 429 HTTP response code if requests are arriving too quickly
 - Revoke the API key if the client violates the usage agreement
- Caution
 - Do not rely exclusively on API keys to protect sensitive, critical, or high-value resource

Restrict HTTP Methods

- Use a whitelist of permitted HTTP methods
 - Only support those on the whitelist
- Reject all requests not on the whitelist with a 405 HTTP response code of “Method not allowed”
- Verify the client is authorized to request the incoming HTTP method on the service

Protect HTTP Methods

- Make sure the incoming request has rights to execute the requested method
- Not everyone should be able to `DELETE` a resource
- Validate the incoming HTTP method
 - Against the session token or API key

Input Validation

- The usual “don’t trust the client” recommendations apply here
- Validate input parameters
- Validate input length, range, and data type
- Reject unexpected or illegal content
- Log validation failures
 - Assume that anyone performing many requests that fail validation rules is not to be trusted

Security Headers

- Send security headers
 - Always send the Content-Type header with the correct type specified
 - This ensures the browser interprets the response correctly
- Send an X-Content-Type-Options: nosniff
 - Make sure the browser does not try to detect a different Content-Type
 - This helps to prevent XSS exploits
- Clients should send an X-Frame-Options: deny
 - Protect against drag and drop clickjacking attacks in older browsers

Protect Against Cross-Site Forgery

- Ensure that all PUT, POST, PATCH, and DELETE requests are protected
 - From cross-site request forgery
- Use a token-based approach
- See this cheat sheet for more detailed information:
 - [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

Cross-Site Request Forgery

```
<!-- This is embedded in another domain's site -->  

```

- Common approach is to use the tag
- This causes the user browser to send a request to a malicious server
- Mitigation techniques include:
 - Short-lived JWTs
 - Special headers added only when they originate from the correct origin
 - Per session cookies
 - Per request tokens
- If JWTs (and session data) are not stored as cookies, CSRF attacks are not possible

HTTP Return Codes

Status code	Message	Description
200	OK	Response to a successful REST API action
201	Created	The request has been fulfilled and resource created. A URI for the created resource is returned in the Location header.
202	Accepted	The request has been accepted for processing, but processing is not yet complete
204	No Content	The request succeeded, but no data is returned in the response
400	Bad Request	The request is malformed, such as message body format error
401	Unauthorized	Wrong or no authentication ID/password provided
403	Forbidden	It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource
404	Not Found	When a non-existent resource is requested
406	Unacceptable	The client presented an unsupported content type in the Accept header
405	Method Not Allowed	The error for an unexpected HTTP method.
413	Payload too large	Use it to signal that the request size exceeded the given size
415	Unsupported Media Type	The requested content type is not supported by the REST service
429	Too Many Requests	The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Techniques for securing RESTful web services
- Session management practices
- Authentication and authorization
- The details of OAuth for authorization

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 5: Cloud Design Patterns

Chapter Overview

In this chapter, we will explore:

- Some challenges in developing applications for cloud deployment
- Some cloud related design patterns
 - Circuit breaker
 - Transparency
 - Health check
 - Service discovery
 - Fail fast
- Creating and deploying an AWS Lambda function

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Challenges in Cloud Development

■ On the following slides, we'll address these topics concerning cloud development:

- Availability
- Data Management
- Messaging
- Management and Monitoring
- Performance and Scalability
- Resiliency
- Security

There are design patterns that address all of these challenges, and many more as well

This is based on information at:

<https://docs.microsoft.com/en-us/azure/architecture/patterns/>

Availability

■ Availability

- The proportion of the time that the system is functional and working properly
- Usually measured as a percentage of uptime

■ Factors that can affect availability include:

- System errors
- Infrastructure problems
- Malicious attacks
- System load

■ Availability is often part of a Service Level Agreement (SLA)

Data Management

- Influences most of the quality attributes of a cloud application
- Data is typically located in multiple locations
 - And across multiple servers
- Data consistency must be maintained
 - Often requiring synchronization across several locations

Messaging

- Cloud applications execute in a distributed environment
- Components and services are loosely coupled for scalability
- Asynchronous communication is widely popular
 - Provides benefits such as scalability
- Asynchronous messaging presents challenges
 - Message ordering
 - Poison message management
 - Poison message: any message that can't be processed
 - Idempotency

Management and Monitoring

- A cloud application executes in a remote data center
- Management and monitoring is more difficult than a local installation
- Services must expose runtime information for administrators and operators

Performance and Scalability

- Performance is the responsiveness of an application to execute a request
- Scalability is the ability to handle increases in load without impacting performance
- Cloud applications experience peaks and valleys in activity
- Cloud applications should be able to respond quickly to activity changes

Resiliency

- Cloud applications should be able to gracefully handle and recover from failures
- There are many sources of potential failures
 - Shared services
 - Competition for resources and bandwidth
 - Remote communications
- Detecting failures and recovering quickly is essential

Security

- Preventing malicious or accidental actions outside of expected usages is important
- Preventing loss or disclosure of sensitive information is essential
- Cloud applications must be designed and deployed to protect them from malicious attacks
 - Restrict access to known, approved users
 - Protect sensitive data



30 min

Exercise 5.1: Researching Cloud Design Patterns

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Bad Things Can and Will Happen

- Many incidents have occurred that verify this maxim
- An airline flight search service is brought to a halt
 - By one method that did not catch an exception
- An online retailer loses massive amounts of revenue
 - Due to some especially costly downtime
- A web application becomes unexpectedly popular
 - And cannot respond to its sudden popularity
 - And a flood of web requests
- Operators for the Three Mile Island reactor:
 - Misinterpreted the meaning of coolant pressure and temperature values
 - Leading them to take exactly the wrong action at every turn

Common Points of Failure for Stability

■ Integration points

- Communication between components
- Subject to network communication problems
- The number one source of problems

■ Blocked threads

- Can happen anytime you check out a resource from a pool
- Or make calls to an external system
- Can cause the system to “hang”

■ Cascading failures

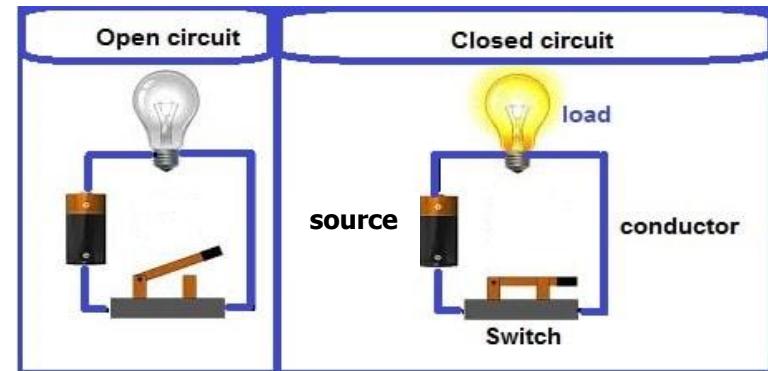
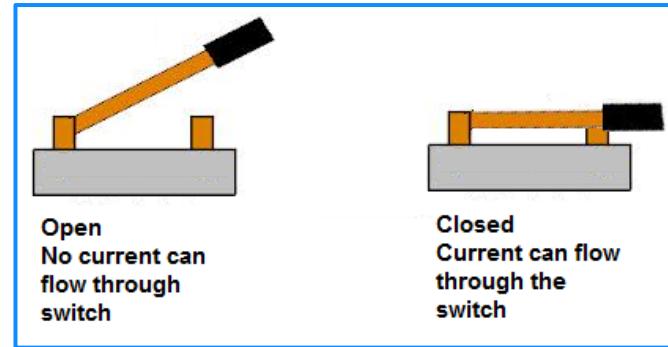
- Common example is a database failure
- A problem in one component may spread to other components
- The problem “jumps the gap”
- The number one accelerator of problems

Stability Solutions

- Circuit breaker
 - Detects when a service is not available
 - Prevents an application from repeatedly trying to call on a failed service
- Transparency
- Health checks
- Service discovery
- Fail fast

Electrical Circuit State Machine

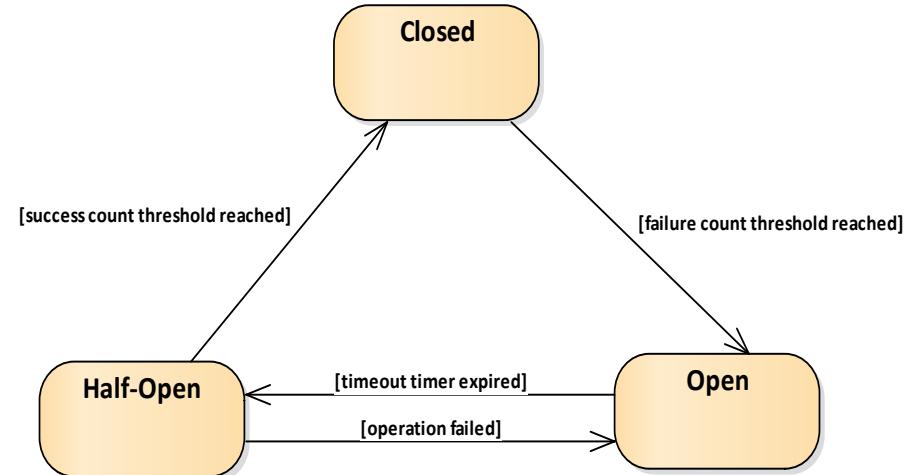
- A circuit breaker is similar to an electrical circuit with a switch
 - The switch determines if the electrical circuit is completed to allow electricity to flow through the circuit and light the bulb
 - The electrical circuit is one of two states
- Closed
 - Electricity flows through the circuit
 - The bulb lights up
- Open
 - The flow of electricity is broken
 - The bulb does not light up



Images from <https://environmentalb.com/electrical-circuits/> and <http://www.learningaboutelectronics.com/Articles/Door-alarm-circuit.php>

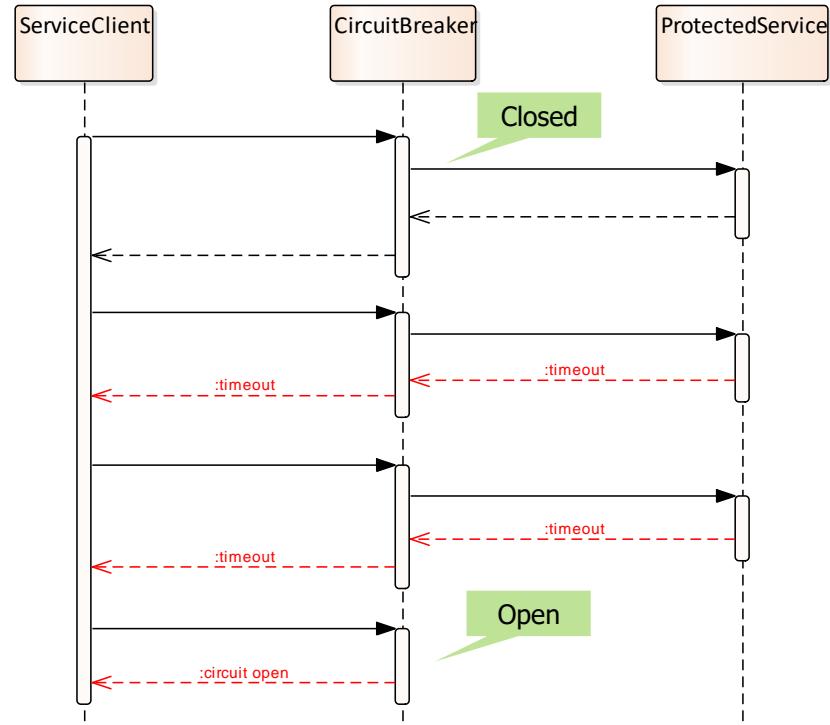
Circuit Breaker State Machine

- A circuit breaker acts as a proxy for the service that may fail
 - It can be modeled with a state machine with the following three states:
- Closed
 - Requests are routed to the service
- Open
 - Requests fail immediately
 - An exception is returned
- Half-Open
 - A limited number of requests are passed to the service to test its availability
 - Prevents the service from being flooded with requests



Circuit Breaker

- One of the most effective patterns to combat cascading failures
- Circuit breaker wraps the component to be protected
 - Monitors it for failures
 - Has a threshold for failed requests
- The circuit breaker will trap calls to a failed service
 - After the threshold is exceeded
 - May utilize a fallback strategy
 - Perhaps route call to a backup service
 - Or return a cached response
- The problem should be reported to ops





HANDS-ON
EXERCISE

Exercise 5.2: Using the Circuit Breaker Pattern

30 min

- Please refer to your Exercise Manual to complete this exercise

Transparency

- Transparency provides information about environmental awareness
 - Historical trends
 - Current system state
 - Such systems will be much easier to debug
- Transparent systems must communicate with the outside world
 - Logging integrated into component source code
 - Make log destinations configurable
 - Use log levels
 - Instance metrics
 - Components should send its metrics to a destination such as a log file
 - Metrics may be difficult to interpret
 - It may take time to learn what is “normal”
 - And what is not

Health Checks

- A health check:
 - Is an application's view of its own health
 - Can help determine what the metrics reported by components actually mean
- What should a health check report?
 - Host ip address
 - Application version
 - Is the component accepting requests?
 - The status of resources
 - Connection pools
 - Caches
 - Circuit breakers

Health Checks (continued)

- The application provides an endpoint (service method) for health monitoring
 - Performs necessary checks
 - Returns an indication of its status
 - The endpoint should require authentication

- A health monitoring system is composed of two factors
 - The checks performed by the application when requested
 - Analysis of the response from the application

Service Discovery

- Service discovery has two parts
 - First, services must be registered somewhere
 - For example, a dynamic pool with load balancer
 - Second, services must be discoverable
 - Clients must know where to send a service discovery request
- Service discovery is a service
 - It can fail
 - Or become overloaded
 - Clients should cache services after discovery
- Don't roll your own discovery service
- There are available implementations
 - Apache Zookeeper
 - HashiCorp's Consul
 - Docker Swarm

Fail Fast

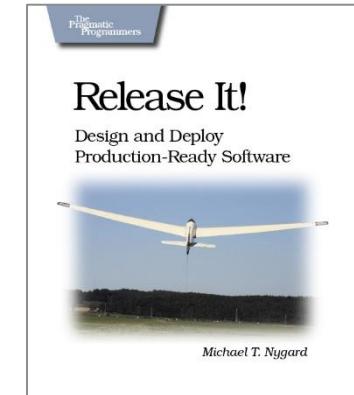
- Slow responses are frustrating
 - Slow failures are even worse!
- If it is possible to determine that an operation will fail:
 - It is better to fail quickly and cleanly
 - Instead of risking arbitrary behavior at some future time
- How to make that determination?
 - Verify any user input before checking on resources
 - Verify all the necessary resources are available before starting to process the request
 - Database connections, circuit breakers, etc.

Fail Fast Guidelines

- Avoid slow responses
 - Don't make users wait for an error message
- Fail fast
 - Don't rely on a timeout to indicate an error condition
- Do basic input validation first
 - If the input is not valid, immediately return an error
- Reserve resources and verify integration points
 - Before starting the response process

Cloud Pattern Resources

- DZone Cloud Design Patterns and Practices
 - <https://dzone.com/articles/cloud-design-patterns-and-practices>
- Microsoft Cloud Design Patterns
 - <https://docs.microsoft.com/en-us/azure/architecture/patterns/>
- CloudAcademy AWS Cloud Design Patterns
 - <https://cloudacademy.com/blog/aws-cloud-design-patterns/>
- Arcitura Education Cloud Computing Design Patterns Catalog
 - <https://patterns.arcitura.com/cloud-computing-patterns>
- AWS Cloud Design Patterns
 - http://en.clouddesignpattern.org/index.php/Main_Page
- *Release It!*, Michael Nygard



Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

AWS Lambda Functions

- AWS Lambda is a cloud technology for *serverless* functions
 - Lets you run code without managing any servers or resources
 - Executes code on demand
 - Scales automatically based on the number of requests
 - Can be written in several different programming languages
- There are several ways in which your code will run
 - Responding to an event
 - Such as modification of data in an Amazon Simple Storage Service (Amazon S3) bucket or a DynamoDB table
 - Responding to an HTTP request
 - For example, a RESTful service call
 - Invoked by using API calls with AWS SDKs
- Serverless applications can be composed of functions
 - Triggered by events

Benefits of AWS Lambda Functions

- Serverless functions or applications
 - There are no servers for you to manage or provision
- Continuous scaling
 - Your application is automatically scaled in response to demand
 - The code runs in parallel
 - Each trigger is processed individually
- Cost efficient
 - You are charged only for the time your code is executing
 - Metered in 100ms increments
 - There is no charge when the code is not running

Lambda Function Reference Architectures

- Web application
 - AWS Lambda functions provide the backend business logic
- Mobile backend
 - AWS Lambda functions run in response to requests from mobile applications
 - Possible functionality:
 - Access to data stored in a DynamoDB database
 - Support users communicating with each other asynchronously
- Real-time stream processing
 - Real-time event data is processed by AWS Lambda functions
 - Possible functionality:
 - Storage of data in a backend datastore
 - Monitoring of aggregate metrics

Core Concepts of Lambda Programming

- Stateless
- Handler
 - The function that Lambda calls to start execution of your Lambda function
- Context
 - A context object is passed to the handler function
- Logging
 - Log entries are written to CloudWatch logs
- Exceptions
 - Used to indicate error conditions
- Concurrency
 - Each instance of your function handles only one request

Lambda Function Implementation

```
public class Greeter {  
    public Map<String, Object> handleRequest(Map<String, Object> request, Context context) {  
        Map<String, String> responseHeaders = Map.of("Content-Type", "application/json");  
        try {  
            String body = String.valueOf(request.get("body"));  
            context.getLogger().log("Request body: " + body);  
  
            Greetings greetings = new ObjectMapper().readValue(body, Greetings.class);  
  
            return Map.of(  
                "headers", responseHeaders,  
                "statusCode", 200,  
                "body", Map.of("message", "Hello " + greetings.getName())  
            );  
        } catch (JsonProcessingException ex) {  
            return Map.of("headers", responseHeaders,  
                "statusCode", 400, "body", Map.of("error", ex));  
        }  
    }  
}
```

AWS passes a Map with details of the HTTP request (URL, data, etc.)

Map item "body" has JSON from HTTP request:
{"name": "Bullwinkle"}

Message logged to Amazon CloudWatch

Converts JSON in request body to a JavaBean

AWS converts returned Map to JSON

Response body:
{"message": "Hello Bullwinkle"}

public class Greetings {
 public String getName() { ... }
 public void setName(String name) { ... }
}

Building and Deploying a Java Lambda Function

- The deployment package can be a zip or JAR file
- To build and deploy with Eclipse and Maven:
 1. Create a new Maven project in Eclipse
 2. Add the `aws-lambda-java-core` dependency to the `pom.xml` file
 3. Add the `maven-shade-plugin` plugin to the `pom.xml` file
 4. Create the Java class and write the code for the Lambda function
 5. Build the project using Maven
 - a. The resulting JAR file can be deployed to AWS Lambda
 - b. Can use the Lambda console or Lambda CLI to upload the Lambda function
- Once the Lambda is deployed, you can assign a unique URL to the function
 - Your Lambda can respond to RESTful HTTP requests on the web
 - You'll experiment with this in the next exercise



30 min

Exercise 5.3: Creating and Deploying a Lambda Function

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Some challenges in developing applications for cloud deployment
- Some cloud related design patterns
 - Circuit breaker
 - Transparency
 - Health check
 - Service discovery
 - Fail fast
- Creating and deploying an AWS Lambda function



Technology Immersion Program

Dynamic Web Application Development

Chapter 6: Node.js

Chapter Overview

In this chapter, we will explore:

- How to use Node.js as your web server
- How to create and use modules
- How to access databases from JavaScript

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

Exporting Functions from Modules

Chapter Summary

What Is Node.js?

- Node.js is a platform built on Google's V8 JavaScript runtime for easily building fast, scalable network applications
- Available for download from <https://nodejs.org/>
- Uses JavaScript, so it allows client- and server-side development with single language
- A “competitor” of Spring Boot for writing server applications
 - If your team has Java developers, use Spring Boot
 - If your team has JavaScript developers, use Node.js

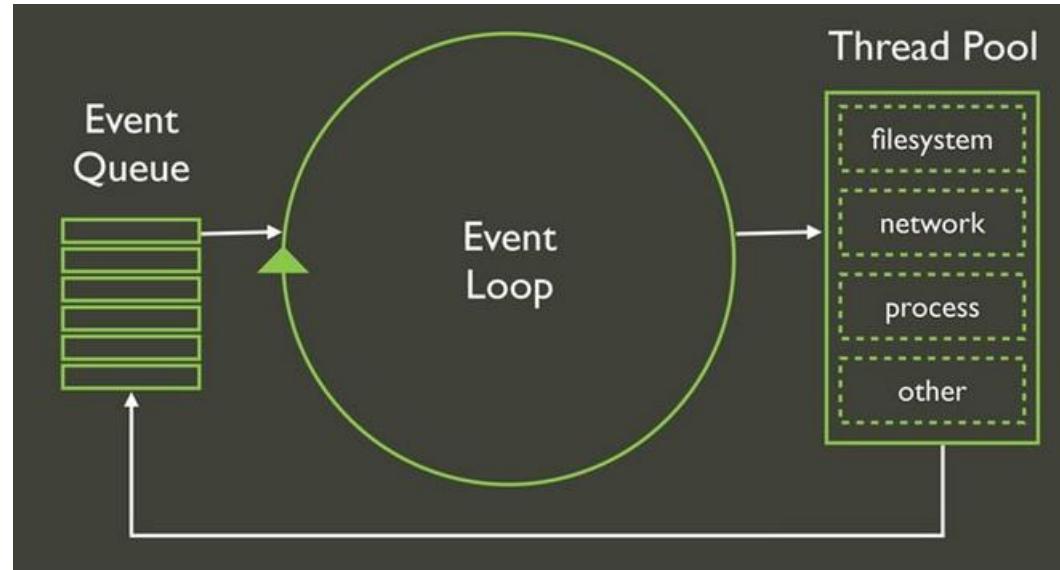


Node.js Features

- Event-driven, lightweight, and efficient with high scalability
- Non-blocking I/O model
 - Asynchronous execution
- High performance—very fast request processing cycle
 - V8 introduced compiled JavaScript
- Supports re-useable modules
- Provides a package manager called “Node Package Manager” (npm)

The Event Loop

- Node.js is an event-driven, single-threaded, non-blocking I/O framework



Blocking Code

- Blocking code forces other JavaScript code in the Node.js process to wait
 - Until the blocking code completes
 - The event loop must wait while the blocking code runs
- The I/O methods in Node.js that are blocking have names that end with `Sync`

```
const fs = require('fs');

// execution blocks on this line until file is read
const data = fs.readFileSync('/file.md', 'utf-8');

console.log(data); // will run after file I/O is complete
moreWork();
```

Non-Blocking Code

- Non-blocking code executes asynchronously
 - All the I/O methods in Node.js provide asynchronous versions
- The asynchronous calls allow higher throughput
 - The callback function executes after the non-blocking code completes

```
const fs = require('fs');
function read_file_cb(err, data) {
    if (err) {
        throw err;
    }
    console.log(data); // may run after moreWork()
}

fs.readFile('/file.md', 'utf-8', read_file_cb);

moreWork(); // may run before console.log() in the callback
```

Getting Started with Node.js

■ Steps to getting started with Node.js:

1. Create a .js file such as `index.js` in a new directory
2. Identify which Node.js “modules” are needed in the code at the top of the file
3. Add your code and save the file
4. Run the following command to run your code:

```
node index.js
```



10 min

Exercise 6.1: Hello World

- Please refer to your Exercise Manual to complete this exercise

What Are Node.js Modules?

- Node.js provides a minimalist core library composed of modules
- Examples of a few built-in “core” modules:

Module	Description	Usage
fs	Provides file I/O	const fs = require('fs');
http	HTTP server and client functionality	const http = require('http');
net	Asynchronous network wrapper	const net = require('net');
path	Utilities for handling and transforming file paths	const path = require('path');
util	Utility functions used by Node.js and custom applications	const utils = require('util');

- Custom modules can be installed using the tool `npm` (more on this later)

Node.js Documentation

- Get documentation on Node.js and the core modules at: <https://nodejs.org/api>

The screenshot shows the Node.js documentation homepage. At the top is the Node.js logo. Below it is a navigation bar with links: HOME, ABOUT, DOWNLOADS, DOCS (which is highlighted in green), FOUNDATION, GET INVOLVED, SECURITY, and NEWS. A large green arrow points down from the DOCS link to the 'About Docs' section. On the left, there's a sidebar with links: Docs (highlighted in green), ES6 in Node.js, FAQ, and API. The main content area is titled 'About Docs'. It contains two paragraphs of text. The first paragraph discusses the different types of documentation available on nodejs.org. The second paragraph provides more detail about the API reference documentation.

Docs

ES6 in Node.js

FAQ

API

About Docs

It's important for Node.js to provide documentation to its users, but documentation means different things to different people. Here, on nodejs.org, you will find three types of documentation, reference documentation, ES6 features, and frequently asked questions.

Our [API reference documentation](#) is meant to provide detailed version information about a given method or pattern in Node.js. From this documentation you should be able to identify what input a method has, the return value of that method, and what, if any, errors may be related to that method. You should also be able to identify which methods are available for different versions of Node.js.

Creating a Node Server

1. The following code creates and runs a server listening on port 8080

```
const http = require('http');

const server = http.createServer( (req, res) => {
    res.end('Hello World from the Server!');
});

server.listen(8080);
```

Server will call this function when it receives an HTTP request

2. To launch the server, open a command prompt and run: `node myFile.js`
3. Navigate to <http://localhost:8080> in your browser to communicate with the server



Exercise 6.2: Creating a Server

10 min

- Please refer to your Exercise Manual to complete this exercise

Creating an HTTP Server to Return HTML

- Node.js can be used to create an HTTP server that returns HTML (or other) content types

```
const http = require('http');

const server = http.createServer( (req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write('<h1>Hello World</h1>');
    res.end();
});

server.listen(8080);
```

Write HTML to the response body



Exercise 6.3: Returning HTML

10 min

- Please refer to your Exercise Manual to complete this exercise

Node.js Modules

- Modules are re-useable/self-contained pieces of software
- Node.js provides a module system for loading application resources:
 - Core Modules – Native modules built-in to Node.js such as http, networking, file system, and more
 - File Modules – Used to load custom modules from .js files
- Packages/Modules can be installed using **npm** – Node Package Manager

Loading Modules

- Modules are loaded by using `require()`
- Core modules are defined using a shortcut name:

```
const http = require('http');
const net = require('fs');
```

- File modules can be loaded by defining a path:

```
const parser = require('./stringParser');
```

No .js at end of file name

Using a Core Node.js Module

- File System module `fs` can be used to read files

```
const fs = require('fs');
const http = require('http');

function create_server_callback(req, res) {
  fs.readFile('myfile.html', (err, fileData) => {
    if (err) {
      console.log(err);
    } else {
      res.write(fileData);
      res.end();
    }
  });
}

const server = http.createServer(create_server_callback);
server.listen(8081);
```

Async callback function

<https://nodejs.org/api/fs.html>



Exercise 6.4: Using a Core Module

10 min

- Please refer to your Exercise Manual to complete this exercise

Creating and Loading a Custom Module

- A custom module “exports” functionality using `module.exports`
 - Exported functions are loaded in other modules using `require()`

hello.js

```
module.exports = function() {  
    return 'Hello!';  
}
```

index.js

```
const greetFunc = require('./hello');  
  
let text = greetFunc();  
console.log(text);
```

Module name is file name
without `.js` extension

Installing Modules with npm

- npm can be used to access packages from <http://npmjs.org>
 - Access thousands of packaged modules
 - Store modules globally or locally in a project
 - Handles dependencies automatically

```
npm help install  
npm install socket.io -g  
npm install underscore  
npm ls
```

Store module in global location

Store module in local
node_modules folder

List local modules

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

Exporting Functions from Modules

Chapter Summary

The package.json File

- Node.js projects have a `package.json` file at their root that defines information such as:
 - Project author
 - Startup scripts
 - Project dependencies
 - Project dev dependencies
- Create a `package.json` file by running `npm init`

```
cmd: npm init

D:\Node_JavaScript\RESTfulServices\Chapter7\SimpleService>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (simpleservice) ▶
```

Creating the package.json File

- npm will prompt for the contents of package.json

```
package name: (simpleservice)
version: (1.0.0)
description: A simple CRUD API for contacts
entry point: (index.js) simple_service.js
test command: jasmine spec
git repository:
keywords:
author: Fidelity FSE
license: (ISC)
About to write to D:\Node_JavaScript\RESTfulServices\  

{
  "name": "simpleservice",
  "version": "1.0.0",
  "description": "A simple CRUD API for contacts",
  "main": "simple_service.js",
  "scripts": {
    "test": "jasmine spec"
  },
  "author": "Fidelity FSE",
  "license": "ISC"
}  
  
Is this OK? (yes) ■
```

package.json File Example

- Dependency version syntax: *majorVersion.minorVersion.patchLevel*

- With no ~ or ^:
all numbers must match exactly
- ~ patch level may be higher
- ^ minor version and/or
patch level may be higher
- To update to latest versions:
 - Run `npm update`
 - `npm` will modify `package.json` with new version numbers

```
{  
  "name": "my-app",  
  "version": "0.1.0",  
  "description": "My super cool node app!",  
  "main": "server.js",  
  "author": "John \"Guru\" Doe",  
  "license": "ISC"  
  "dependencies": {  
    "rxjs": "~6.5.3",  
    "socket.io": "^1.3.5"  
  },  
  "devDependencies": {  
    "gulp": "^3.8.11"  
  }  
}
```

Installing Modules into package.json

- The `npm install` command installs packages and updates `package.json`:

`--save`

`--save-dev`

```
npm install socket.io --save  
npm install gulp --save-dev
```

Install module and update
dependencies property

Install module and update
devDependencies property

Note:

As of **npm** version 5,
`--save/-/--save-prod`"
are the defaults, so
`"npm install <module>"`
saves dependency to
`package.json` by default

Note:

- `devDependencies` are for the development-related scripts, e.g., unit testing, packaging scripts, documentation generation, etc.
- `dependencies` are required for production use, and assumed required for dev as well



Exercise 6.5: Using npm

10 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

Exporting Functions from Modules

Chapter Summary

Database and Node.js

- Most database have easy to install processes (e.g., npm install, etc.), installing the necessary drivers and modules
 - E.g., for MS SQL Server: `npm install mssql`
- Oracle and Node.js
 - The `node-oracledb` add-on for Node.js supports high-performance Oracle applications
 - May require separate installation of Oracle client libraries
 - Installation instructions:
https://node-oracledb.readthedocs.io/en/latest/user_guide/installation.html
- You'll use `oracledb` later in this course to interact with an Oracle database

Custom Approach

- Each database (MySQL, MSSQL, Oracle, etc.) has its own approach
- Research the Internet and sufficient code examples will be available
- *Note:* Don't forget to run the `npm install` from within your express project or the modules won't be found!

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

Exporting Functions from Modules

Chapter Summary

Exporting Functions

- JavaScript supports two ways to export functions from a module

- Assign a reference to a single function directly to the global `module.exports` object

hello.js

```
module.exports = () => 'Hello!';
```

index.js

```
const sayHi = require('./hello');  
console.log(sayHi());
```

Call the only function
exported by module

- Client module can choose any name for the function

- Assign references to multiple functions by adding properties to `module.exports`
 - Every accessible function must be defined as `module.exports.functionname`

multi.js

```
module.exports.list = () => 'List';  
module.exports.other = () => 'Other';
```

Call one of the functions exported by the module

index.js

```
const multiMod = require('./multi');  
console.log(multiMod.list()); // List  
console.log(multiMod.other()); // Other
```

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

Exporting Functions from Modules

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to use Node.js as your web server
- How to create and use modules
- How to access databases from JavaScript

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 7: Node.js and Express

Chapter Overview

In this chapter, we will explore:

- How to debug a Node.js application
- How to utilize Express to implement RESTful APIs with Node.js
- How to test a RESTful API with Jasmine

Chapter Concepts

Debugging a Node Application

Defining a RESTful API with Express

CORS (Cross-Origin Resource Sharing)

Testing RESTful APIs with Jasmine

Chapter Summary

Debugging Node.js

- Tools for debugging Node.js programs:
 - **Node Debug:** a command-line debugging utility
 - <http://nodejs.org/api/debugger.html>
 - **Chrome Developer Tools:** can debug server-side JavaScript as well as client-side
 - **Visual Studio Code**
- We will focus on Visual Studio Code

Debugging with Visual Studio Code

The screenshot shows the Visual Studio Code interface in debug mode. The left sidebar contains the 'Variables' view (with a 'Block' section showing 'temp: 4'), 'Watch' view (empty), 'Call Stack' view ('PAUSED ON BREAKPOINT' at 'calc_pi' in 'pi.js' at line 7:9), and 'Breakpoints' view ('All Exceptions' and 'Uncaught Exceptions' sections). The main editor area displays a JavaScript file named 'pi.js' with the following code:

```
1 function calc_pi() {
2     let n = 200000;
3     let pi = 0;
4     for (var i = 0; i < n; i++) {
5         let temp = 4 / (i * 2 + 1);
6         if (i % 2 == 0) {
7             pi += temp;
8         }
9         else {
10            pi -= temp;
11        }
12    }
13    return pi;
14}
15 let pi = calc_pi();
16 console.log(pi);
17
```

The line 'pi += temp;' is highlighted with a yellow background, indicating it is the current line of execution. The bottom status bar shows the file path 'pi.js - CourseDev - Visual Studio ...', line 'Ln 7, Col 9', and encoding 'UTF-8'. The bottom right corner includes a 'Go Live' button and a bell icon.



HANDS-ON
EXERCISE

Exercise 7.1: Debugging Node.js

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Debugging a Node Application

Defining a RESTful API with Express

CORS (Cross-Origin Resource Sharing)

Testing RESTful APIs with Jasmine

Chapter Summary

Getting Started with Express

- The Express module provides a framework for implementing RESTful APIs
- Get started with Express by installing Express globally with **npm**:

```
npm install express -g
```

A Node.js RESTful API

- We'll create a Node.js version of the Widget/Gadget CRUD RESTful API
- First task: write functions that handle HTTP requests
 - In Version 1, CRUD functions get data from a mock DAO
- All functions take two parameters:
 - req – HTTP request
 - res – HTTP response
- getAllWidgets() – returns a list of widgets as JSON

```
rest_controller.js
```

```
const ProductDao = require('../dao/mock-product-dao');
const productDao = new ProductDao();

function getAllWidgets(req, res) {
  try {
    const widgets = productDao.queryForAllWidgets();
    res.json(widgets);
  } catch (err) {
    console.log(`error GET widgets: ${err}`);
    res.status(500).json({ error: err });
  }
}
```

Get data from DAO

Send JSON response with default status 200

Send JSON response with status 500

```
[{"id": 1, "description": "...", ...}, {"id": 2, "description": ...}, {...}]
```

Defining Routes in Express

- Next task: configure Express server to route REST requests to the appropriate handler method
- Use Express's Router class to create *route handlers*
 - Route handler maps HTTP requests to your functions

```
rest_controller.js  
const express = require('express');  
  
const app = express();  
  
app.use(express.json());  
  
router = express.Router();  
  
router.get('/widgets', getAllWidgets);  
  
app.use('/', router);  
  
app.listen(3000,  
() => console.log( 'Listening on port 3000'))
```

rest_controller.js

Create an instance of Express

Configure Express to parse JSON in request body

Create a route handler

Handler will map GET request to your function

Express will use your router mappings for requests

Start the app on the given port

Sending Requests to the RESTful API

- Open a terminal in Chapter7/examples/RestControllerGlobalFunctions
 - Run these commands:

```
npm install express  
npm start
```

- Use Insomnia to send a GET request to `http://localhost:3000/widgets`

The screenshot shows the Insomnia REST Client interface. The top navigation bar has 'No Environment' and 'Cookies'. The main header shows a 'GET' request to 'http://localhost:3000/widgets'. Below the header are tabs for 'Body', 'Auth', 'Query', 'Headers', and 'Docs'. A status summary shows '200 OK', '7.52 ms', '234 B', and '1 Minute Ago'. Below the status is a 'Preview' tab which is active, showing a JSON response. The response data is:

```
1 [  
2 {  
3   "id": 1,  
4   "description": "Good Widget",  
5   "price": 12.99,  
6   "gears": 2,  
7   "sprockets": 3  
8 }]
```

Defining Other CRUD Operations

- Now we'll define functions and routes for the remaining CRUD operations

```
...
router = express.Router();
router.get('/widgets', getAllWidgets);
router.get('/widgets/:id', getWidget);
router.post('/widgets', addWidget);
router.put('/widgets', updateWidget);
router.delete('/widgets/:id', deleteWidget);

app.use('/', router);

function getAllWidgets(req, res) { ... }
function getWidget(req, res) { ... }
function addWidget(req, res) { ... }
function updateWidget(req, res) { ... }
function deleteWidget(req, res) { ... }
```

GET and DELETE requests
retrieve inputs from URL

POST and PUT retrieve inputs
from URL or request body

Handling a GET Request with a Parameter

- Use Express route parameters to capture segments of the request URL

```
router.get('/widgets/:id', getWidget);
```

- In your handler function, use the request's `params` property to get the parameter value

```
GET http://localhost:3000/widgets/3
```

```
function getWidget(req, res) {  
  const id = req.params.id; // Get path parameter from params  
  
  try {  
    const widget = productDao.queryForWidget(id);  
    if (widget) {  
      res.json(widget);  
    } else {  
      res.status(400).json({ error: `Widget ${id} not found` });  
    }  
  } catch (err) { ... } // Set status 400 (Bad Request)  
}  
Use the parameter value
```

Handling a POST or PUT Request

- JSON in body of POST or PUT request is automatically converted to a JavaScript Object

```
function addWidget(req, res) {  
    const w = req.body; // Get JavaScript object with request data  
  
    if (w && w.id && w.description && w.price && w.gears && w.sprockets) {  
        try {  
            const insertCount = productDao.createWidget(w);  
            res.json({ rowCount: insertCount }); // Convert a JavaScript object to a JSON string  
        }  
        catch (err) {  
            res.status(500).json({ error: 'error on POST request' }); // Return status 500  
        }  
    }  
    else {  
        res.status(400).json({ error: 'widget is not fully populated: ' +  
            JSON.stringify(w) }); // Populate the response body with JSON  
    }  
}
```

JSON as Data File Format

- If no database is available at development time, use a mock data source
 - Define widget data in a JSON file
 - Read the JSON file using standard Node.js file functions

widgets.json

```
[  
  { "id": 1, "description": "Good Widget", "price": 12.99, "gears": 2, "sprockets": 3 },  
  { "id": 2, "description": "Better Widget", "price": 42.99, "gears": 5, "sprockets": 5 },  
  { "id": 3, "description": "Best Ever Widget", "price": 89.99, "gears": 10, "sprockets": 8 }  
]
```

- Define a stub DAO class that reads the parsed JSON data
 - See example on next slide

Define a Mock DAO

```
const fs = require('fs');
const widgetFilePath = `__dirname/widgets.json`;

class ProductDao {
    constructor() {
        const fileContents = fs.readFileSync(widgetFilePath, 'utf-8');
        this.widgets = JSON.parse(fileContents);
    }

    queryForAllWidgets() {
        return widgets;
    }

    queryForWidget(id) {
        const searchResult = widgets.filter(w => w.id === id);
        const widget = searchResult ? searchResult[0] : null;
        return widget;
    }

    createWidget(widget) {
        widgets.push(widget);
        return 1;
    }
}
```

Global variable with current directory path

Parsing the file yields an array of JavaScript objects

Return the array of mock widgets

Search for the widget with the given ID

Add the widget to the array of widgets

mock_product_dao.js

Defining a RESTful Controller with a Class

- To make our RESTful controller easier to test, we'll convert it to a class
 - Test cases will replace the controller's DAO property with a mock DAO

```
class ProductRestController {
  constructor() {
    this.port = process.env.PORT || 3000;
    this.productDao = new ProductDao();           Create a ProductDao
    this.app = express();
    this.app.use(express.json());                 Configure Express as before
    ...
  }

  start() {
    this.app.listen(this.port, () => console.log(`listening on ${this.port}`))
  }
}
```

Binding Methods to an Object

- Replace global handler functions with methods in the controller class
 - Unlike global functions, methods always have a reference to the current object: `this`
 - In the route configuration, use JavaScript's `bind()` function to assign a value to `this`

```
class ProductRestController {  
  ...  
  constructor() {  
    ...  
    router.get('/widgets', this.getAllWidgets.bind(this));  
  }  
  
  getAllWidgets(req, res) {  
    const widgets = await this.productDao.queryForAllWidgets();  
  
    res.json(widgets);  
  }  
}
```

When `getAllWidgets` is called ...

... bind its `this` reference to the current object (i.e., the controller)

Starting the API Server

- Within every Node module, the global variable `module` references the current module
 - Also, Node sets the global variable `require.main` to the program being executed
 - So, if `require.main === module`, this file is being executed as a standalone program
 - Otherwise, it is being “required” by another module; e.g., a spec module

```
class ProductRestController {  
    ...  
    start() {  
        this.app.listen(this.port, () => console.log(`listening on ${this.port}`))  
    }  
}  
  
module.exports = ProductRestController;  
  
if (require.main === module) {  
    const controller = new ProductRestController();  
    controller.start();  
}
```

rest_controller.js

Guard condition: is this file being executed as a program?

If so, call the controller's `start()` method



30 min

Exercise 7.2: Creating a RESTful API with Express

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Debugging a Node Application

Defining a RESTful API with Express

CORS (Cross-Origin Resource Sharing)

Testing RESTful APIs with Jasmine

Chapter Summary

What Is CORS?

- Security is a primary concern for all web applications
 - RESTful APIs need to follow the same security best practices as other web applications
- Cross-domain requests present a serious security exposure
 - JavaScript downloaded from one host sends requests to a different host
 - Modern browsers forbid cross-domain requests by default
 - But many web applications need this capability
- Cross-Origin Resource Sharing (CORS) permits control over cross-domain requests
 - Your application configures which hosts it will send requests to
 - Client's browser enforces your CORS policy

Implementing CORS

- CORS is implemented with standard HTTP headers in requests
- Client sends CORS HTTP request headers to the server before sending the actual request:
 - `Origin`: Hostname and port number that caused the request
 - `Access-Control-Request-Method`: HTTP method client will use for actual request
 - `Access-Control-Request-Headers`: HTTP headers client will include in actual request
- Handling these headers manually is a difficult task
- Use the `cors` npm package to enable CORS in an Express application
 - `npm install cors`

CORS: Whole Service or One Route Only?

- After npm installs the package, add it to your declaration list:

```
const cors = require('cors');
```

- For application-wide use, add the following:

```
app.use(cors());
```

WARNING! This opens your application to malicious sites making unauthorized requests!

- Best practice: enable CORS for specific routes only

```
router.get('/widgets', cors(), getWidgets);
```

Chapter Concepts

Debugging a Node Application

Defining a RESTful API with Express

CORS (Cross-Origin Resource Sharing)

Testing RESTful APIs with Jasmine

Chapter Summary

Setup for Testing with Jasmine

- Let's add Express and Jasmine to our project
 - npm install express
 - npm install jasmine --save-dev
- Create a Jasmine configuration file spec/support/jasmine.json
 - npx jasmine init
- For more details, documentation, and examples, visit the following websites:
 - <https://jasmine.github.io/>
 - <https://angular-university.io/lesson/angular-testing-jasmine-spies>
 - <https://howtodoinjava.com/javascript/jasmine-unit-testing-tutorial/>

Setup for Testing with Jasmine (continued)

- The Jasmine installation adds a test script to your package.json file

```
package.json  
"main": "src/rest-controller/rest-controller.js",  
"scripts": {  
  "test": "jasmine",  
  ...  
}
```

Jasmine Review

- describe() defines a suite of specs (i.e., unit tests)
- Nested calls to it() define the specs themselves

```
describe('widget constructor', () => {
  it('succeeds when all args are valid', () => {
    const widget = new Widget(1, 'A widget', 9.99, 5, 3);

    expect(widget.id).toBe(1);
    expect(widget.description).toBe('A widget');
    expect(widget.price).toBe(9.99);
    expect(widget.gears).toBe(5);
    expect(widget.sprockets).toBe(3);
  });

  it('fails when an arg is invalid', () => {
    expect(() => new Widget(1, 'A widget', 9.99, 5, 0)).toThrowError(Error);
  });
});
```

```
class Widget extends Product {
  constructor(id, description, price, gears, sprockets) {
    super(id, description, price);
    if (gears <= 0 || sprockets <= 0) {
      throw Error(`invalid arg ${gears} or ${sprockets}`);
    }
    this.gears = gears;
    this.sprockets = sprockets;
  }
}
```

widget.spec.js

Assert object's properties have correct values

Assert constructor threw an error

Jasmine Matchers

■ Jasmine defines *matcher* methods that are chained to `expect()`

■ You can disable a suite or spec

- Replace `describe()` with `xdescribe()`
- Replace `it()` with `xit()`

■ You can run one suite only or one spec only

- Replace `describe()` with `fdescribe()` ("focused")
- Replace `it()` with `fit()`

<code>toBe()</code>	passed if the actual value is of the same type and value as that of the expected value. It compares with <code>==</code> operator
<code>toEqual()</code>	works for simple literals and variables; should work for objects too
<code>toMatch()</code>	to check whether a value matches a string or a regular expression
<code>toBeDefined()</code>	to ensure that a property or a value is defined
<code>toBeUndefined()</code>	to ensure that a property or a value is undefined
<code>toBeNull()</code>	to ensure that a property or a value is null.
<code>toBeTruthy()</code>	to ensure that a property or a value is <code>true</code>
<code>ToBeFalsy()</code>	to ensure that a property or a value is <code>false</code>
<code>toContain()</code>	to check whether a string or array contains a substring or an item.
<code>toBeLessThan()</code>	for mathematical comparisons of less than
<code>toBeGreaterThan()</code>	for mathematical comparisons of greater than
<code>toBeCloseTo()</code>	for precision math comparison
<code>toThrow()</code>	for testing if a function throws an exception
<code>toThrowError()</code>	for testing a <i>specific</i> thrown exception

Spies – Why Do We Need Them?



- The Widget constructor is easy to test; it changes the state of an object
 - Our specs simply call the method and examine the object's new state
- Problem: a REST controller is harder to test
 - Controllers are stateless, and their methods don't return values
 - How can we tell if a method succeeded?
- Solution: create Jasmine *spies* to test the controller
- Jasmine spies have methods with these capabilities:
 1. Can be configured to return a different value or throw a different error for each call, and
 2. Will record how their methods were called
 - How many calls
 - Argument values
- Spies make it easy to thoroughly test all positive and negative scenarios

Spies – Why Do We Need Them? (continued)



- What spies are required to test our controller's `getAllWidgets()` method?
 - Positive test: mock DAO will return a valid array of widgets
 - Negative test: mock DAO will throw an error
- In both cases, we also need a mock HTTP response to tell us which methods were called

```
class ProductRestController {  
    getAllWidgets(req, res) {  
        try {  
            const widgets = this.productDao.queryForAllWidgets();  
  
            res.json(widgets);  
        } catch (err) {  
            res.status(500).json({error: err});  
        }  
    }  
}
```

We need a mock DAO that either returns a list of widgets or throws an error

We need a mock HTTP response that records which methods were called and their argument values

Unit Test for RESTful Controller

- In `beforeEach()`, create a controller instance, mock DAO, and mock HTTP response

```
const testWidgets = [{ id: 1, description: 'Low Impact Widget', ... }, { ... }, { ... }];  
  
describe('RESTful controller unit tests for Widget CRUD operations:', () => {  
    let controller;  
    let mockDao;  
    let mockHttpResponse;  
  
    beforeEach(() => {  
        mockDao = jasmine.createSpyObj('mockDao', ['queryForAllWidgets', 'queryForWidget',  
            'createWidget', 'updateWidget', 'deleteWidget', 'shutdown']);  
  
        controller = new ProductRestController();  
        controller.productDao = mockDao;  
  
        mockHttpResponse = jasmine.createSpyObj('mockHttpResponse', ['status', 'json']);  
  
        // The mock status() method needs to return a reference to the mock response  
        // so it can be chained with other calls: res.status(500).json(...)  
        mockHttpResponse.status.and.returnValue(mockHttpResponse);  
    });  
});
```

Create mock object with these methods

Names must match method names of real DAO

"Inject" mock DAO into controller

Configure return value of mock `status()` method

Unit Test for RESTful Controller (continued)

Configure the mocks for positive and negative tests

```
describe('retrieve all widgets', () => {
  it('succeeds', () => {
    mockDao.queryForAllWidgets.and.returnValue(testWidgets);
    const req = { };
    controller.getAllWidgets(req, mockHttpResponse);
    expect(mockHttpResponse.json)
      .toHaveBeenCalledWith(testWidgets);
  });
  it('fails due to a DAO exception', () => {
    mockDao.queryForAllWidgets.and.throwError('error');
    const req = { };
    controller.getAllWidgets(req, mockHttpResponse);
    expect(mockHttpResponse.status).toHaveBeenCalledWith(500);
  });
});
```

Configure mock `queryForAllWidgets()` to return the array of test widgets

```
getAllWidgets(req, res) {
  try {
    const widgets = this.productDao.queryForAllWidgets();
    res.json(widgets);
  } catch (err) {
    res.status(500).json({error: err});
  }
}
```

Call method under test

rest-controller.js

Configure mock `queryForAllWidgets()` to throw an error

Verify mock `status()` was called with argument 500

Unit Test for POST Request

- Unit test for the controller method that handles a POST request to add a widget

```
it('succeeds', () => {
  mockDao.createWidget.and.returnValue(1);
  const req = {
    body: { id: 99, description: 'Widget', ... }
  };
  controller.addWidget(req, mockHttpResponse);

  expect(mockDao.createWidget).toHaveBeenCalled();
  expect(mockHttpResponse.json).toHaveBeenCalledWith({ rowCount: 1 });
});

it('fails due to an empty request body', () => {
  const req = { body: {} };

  controller.addWidget(req, mockHttpResponse);

  expect(mockHttpResponse.status).toHaveBeenCalledWith(400);
  expect(mockDao.createWidget).not.toHaveBeenCalled();
});
```

```
addWidget(req, res) {
  const w = req.body;
  if (w && w.id && w.description && w.price) {
    const rowCount = this.productDao.createWidget(w);
    res.json({ rowCount: rowCount });
  } else {
    res.status(400).json({error: `invalid ${w}`});
  }
}
```

Fake request with valid body

Fake request with empty body

rest-controller.js



Exercise 7.3: Testing a RESTful API with Jasmine

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Debugging a Node Application

Defining a RESTful API with Express

CORS (Cross-Origin Resource Sharing)

Testing RESTful APIs with Jasmine

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to debug a Node.js application
- How to utilize Express to implement RESTful APIs with Node.js
- How to test a RESTful API with Jasmine

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 8: Promises and Testing Node with Jasmine

Chapter Overview

At the end of this chapter, you will be able to:

- Work with asynchronous JavaScript functions that return Promises
- Simplify asynchronous code with `async` and `await`
- Test asynchronous RESTful methods using the Jasmine framework
- Perform CRUD operations on an Oracle database using Node.js

Chapter Concepts

Callbacks and Promises

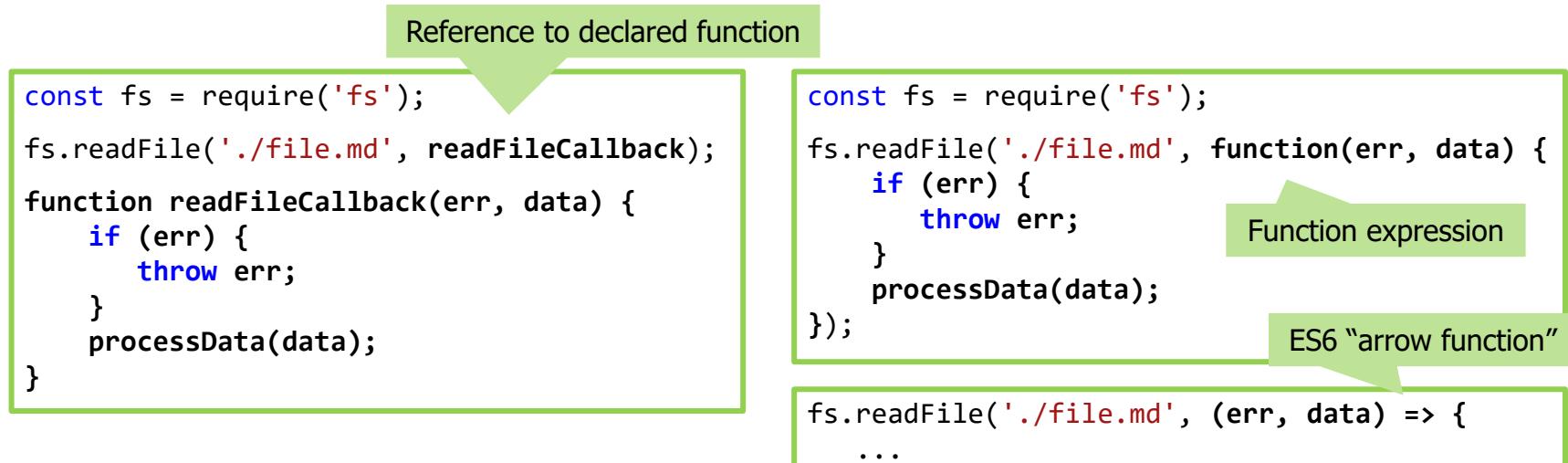
Accessing Oracle with Node.js

Testing Asynchronous Functions

Chapter Summary

Callbacks

- JavaScript library functions often take a *callback* argument
 - Callback: a function passed as an argument to another function
 - Example: in Node standard library, `fs.readFile()` takes a callback argument
 - When the read operation is complete, `readFile()` calls the callback
 - Callback can be a reference to a declared function, or a function expression:



Callbacks with Asynchronous Operations

- Callbacks are good for asynchronous operations
 - When the order of processing is unpredictable
- Example: handling button clicks in a jQuery UI

```
$('#previous_button').click(() => {
  ... // handle "Previous" button click
});

$('#play_button').click(() => {
  ... // handle "Play" button click
});

$('#next_button').click(() => {
  ... // handle "Next" button click
});
```

Argument to `click()` method is a callback function

User may click buttons in any order, so callback functions may be called in any order

Callbacks with Synchronous Operations

- **But:** callbacks aren't good for operations that must execute in a specific order
- Example: Reading transactions from a file to update records in a database
- Scenario:
 1. Send web service request to get current record from database
 2. Update record with new transaction data from input file
 3. Send web service request with updated record
- All steps require calls to asynchronous functions
 - But each step must complete before the next step begins
 - The asynchronous results must be processed in the correct order
- Implementing this use case with traditional callbacks results in "Callback Hell"
 - Deeply nested callbacks make code difficult to understand and maintain
 - See example on next slide

Callback Hell

```
function updateRecordFromTxnFile(filepath, txnId) {
    getRecordFromWebService(txnId, function(err, currentRecord) {
        if (err) {
            failureCallback(err);
        }
        else {
            updateRecordFromTxnFile(filepath, currentRecord, function(err, updatedRecord) {
                if (err) {
                    failureCallback(err);
                }
                else {
                    sendUpdatedRecord(updatedRecord, function(err, confNum) {
                        if (err) {
                            failureCallback(err);
                        }
                        else {
                            console.log(`Confirmation number ${confNum}`);
                        }
                    });
                }
            });
        }
    });
}
```

What's Really Going On?

```
function updateRecordFromTxnFile(filepath, txnid) {  
    getRecordFromWebService(txnid, function(err, currentRecord) {  
        if (err) {  
            failureCallback(err);  
        }  
        else {  
            updateRecordFromTxnFile(filepath, currentRecord, function(err, updatedRecord) {  
                if (err) {  
                    failureCallback(err);  
                }  
                else {  
                    sendUpdatedRecord(updatedRecord, function(err, confNum) {  
                        if (err) {  
                            failureCallback(err);  
                        }  
                        else {  
                            console.log(`Confirmation number ${confNum}`);  
                        }  
                    });  
                }  
            });  
        }  
    });  
}
```

1. Call getRecordFromWebService() ...

2. ... then call updateRecordFromTxnFile() ...

3. ... then call sendUpdatedRecord() ...

4. ... then log the final result

... but if there's an error anywhere,
call failureCallback()

Promises: Structured Callbacks

- *Promise*: object that represents the eventual result of an asynchronous operation
 - Allows asynchronous methods to return values like synchronous methods
 - Asynchronous method returns a *promise* to supply the value at some point in the future
- A Promise has two main methods:
 - `then(successAction)` – *successAction* is a function to be executed on success
 - `catch(errorAction)` – *errorAction* is a function to be executed on error
- `then()` and `catch()` both return Promises
 - Permits chaining of method calls

```
function updateRecordFromTxnFile(filepath, txnId) {  
    getRecordFromWebService(txnId)  
        .then(currentRecord => updateRecordFromTxnFile(filepath, currentRecord))  
        .then(updatedRecord => sendUpdatedRecord(updatedRecord)))  
        .then(confNum => console.log(`Confirmation number ${confNum}`))  
        .catch(err => failureCallback(err));  
}
```

If functions from previous example return Promises, code is much simpler

Standard Library Usage Without Promises

- Many standard Node modules now have a Promise-based API
 - Example: global replacement of a string in a file using `fs` module
 - Code below uses “classic” nested callback technique; code on next slide uses Promises

```
const fs = require('fs'); // require "classic" fs module  
function replaceInFileCallback(filename, str, repl) {  
    fs.readFile(filename, 'utf8', (err, contents) => { // pass callback to fs.readFile()  
        if (err) {  
            console.error(err);  
        } else {  
            let result = contents.toString().replace(new RegExp(str, 'g'), repl); // do the replacement  
            fs.writeFile(filename, result, 'utf8', (err) => { // pass callback to fs.writeFile()  
                if (err) {  
                    console.error(err);  
                } else {  
                    console.log(`done replacing ${str} with ${repl} in ${filename}`);  
                }  
            });  
        }  
    }  
}
```

`replace_in_file_callbacks.js`

Standard Library Usage with Promises

Version 2: using `fs.promises` module for file operations

- We still need callbacks as arguments to `then()` and `catch()`
- But callbacks won't be nested

```
const fs = require('fs').promises; // require fs Promise API          replace_in_file_promises.js

function replaceInFile(filepath, str, repl) {
  return fs.readFile(filepath) // Read file asynchronously and return a Promise
    .then(contents => { // Result of read operation is input to next callback
      let result = contents.toString().replace(new RegExp(str, 'g'), repl);
      return fs.writeFile(filepath, result); // Write asynchronously, return a Promise
    })
    .then(() => console.log(`done replacing ${str} with ${repl} in ${filepath}`))
    .catch(err => console.log(err));
}
```

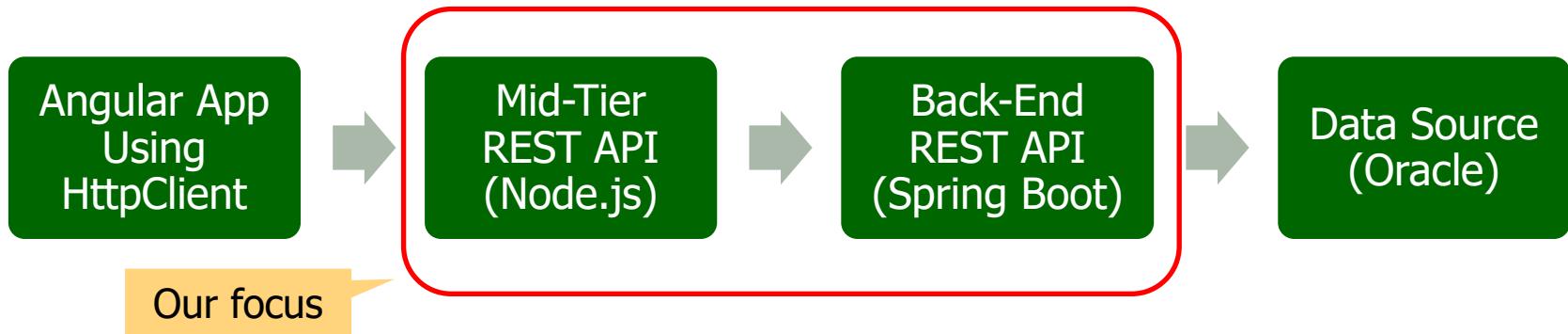
Promise-Based HTTP Client for Node.js: Axios

- Axios is designed to perform various HTTP-based requests such as GET/POST/DELETE and others
- In addition, the following list summarizes Axios' capabilities:
 - Make [XMLHttpRequests](#) from the browser
 - Make [HTTP](#) requests from Node.js
 - Supports the [Promise](#) API
 - Intercept request and response
 - Transform request and response data
 - Cancel requests
 - Automatic transforms for JSON data
 - Client-side support for protecting against [XSRF](#)

A X I O S

Introducing a Mid-Tier RESTful Service

- Currently our Angular frontend communicates directly with the back-end API
 - In production, client code and API are deployed on different hosts
 - This is a cross-origin request: requires CORS to be enabled
- To eliminate this security risk, we add a lightweight mid-tier RESTful service layer
 - Mid-tier service is deployed on same host as client: no need for CORS
 - Client now sends requests to mid-tier service
 - Mid-tier service forwards requests to back-end API



Using Axios to Get Data From a Back-End Service

- Our mid-tier service can use Axios to make a GET request to the back-end service
 - Install Axios: `npm install axios`

```
var axios = require('axios');

router.get('/fetch', (request, response) => {
  axios.get('http://localhost:8080/widgets')
    .then(backEndResp => {
      console.log(`back-end service returned ${backEndResp.data}`);
      response.json(backEndResp.data);
    })
    .catch((err) => {
      console.error(`back-end service error: ${err}`);
      response.status(500).send({error: err});
    });
});
```

Axios throws an error if response status != 2xx

Loading Axios

Make simple GET request. Headers are not needed. JSON is default data format.

catch callback is executed only when a Promise is rejected

Error management is a best practice

The `async` and `await` Keywords

- Promises clean up “callback hell” considerably
 - But you still must pass callbacks to a Promise’s `then()` and `catch()`
- ECMAScript 2017 (ES8) introduced `async` and `await`
 - Makes complex asynchronous code look more like simple synchronous code
- `async` – tells JavaScript a function will always return a Promise

```
async function readTransactions() { ... }
```
- If necessary, JavaScript will wrap the function’s return value in a Promise
 - The Promise will resolve to the function’s return value
- `await` on a function call hides the Promise returned from a function

```
const content = await readTransactions();
```
- With `await`, you access the return value exactly as with a synchronous function
 - Note: `await` can be used only within functions defined as `async`

The `async` and `await` Keywords (continued)

Instead of this:

```
function readTransactions() {  
    readData('./data/txn.json')  
        .then(contents => writeData('./data/txn-copy.json', contents))  
        .then(() => console.log('file copied successfully'))  
        .catch(err => console.log(`file copy failed: ${err}`));  
}
```

`readData()` and
`writeData()` return Promises

Do this:

Function must be declared `async`

You can ignore the returned
Promise if you don't need its value

Catch exceptions to avoid
`UnhandledPromiseRejection`

```
async function readTransactions() {  
    try {  
        const contents = await readData('./data/txn.json');  
        await writeData('./data/txn-copy.json', contents);  
        console.log('file copied successfully');  
    }  
    catch (err) {  
        console.log(`file copy failed: ${err}`);  
    }  
}
```

Assign the Promise's
resolved value to a variable

Posting Data to a Back-End Service

Making a POST request with `async` and `await`

```
router.get('/add/42', async (request, response) => {
  const result = await doPostRequest();
  response.json( { title: 'Made a POST call', data: result } );
});
```

Callback must be `async` because it uses `await`

```
async function doPostRequest() {
  const bodyData = { "id": "42", "description": "Hypersonic Widget", ... };
  try {
    const backEndResp = await axios.post('http://localhost:8080/widgets', bodyData);
    return backEndResp.data;
  }
  catch (err) {
    console.log('Caught error posting data');
    throw Error('something went wrong!: ' + err);
  }
}
```

Calling asynchronous `doPostRequest()` to perform `axios.post()`

This function will return a Promise

A hard-coded new dataset

`await` handles the Promise returned by `axios.post()`

Catch errors with `try/catch`

Exception thrown from `async` function is automatically wrapped in a rejected Promise

Handling Axios Errors when Fetching Resources

- If our back-end contact API can't find a contact with the specified ID, it returns status 406
 - Axios throws an error if an HTTP request returns a 4xx or 5xx status
 - So in our mid-tier code, we need to test for the 406 status in a `catch` block

```
async getContact(req, res) {  
    try {  
        const id = req.params.id;  
        const backEndResp = await axios.get('http://.../' + id);  
        res.json(backEndResp.data);  
    }  
    catch (err) {  
        if (err.response.status == 406) {  
            res.status(406).json({ error: `Contact ${id} not found` });  
        }  
        else {  
            res.status(500).json({error: err});  
        }  
    }  
}
```

If back-end returns status 406, Axios throws an error

If back-end returns status 200, we return the data

Test for the 406 here in the `catch`, not in the `try`

Handle other error statuses



Exercise 8.1: Using JavaScript Promises

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Callbacks and Promises

Accessing Oracle with Node.js

Testing Asynchronous Functions

Chapter Summary

Node.js and Oracle

- node-oracle is an open-source Node.js add-on maintained by Oracle
 - Documentation and examples:
 - <https://www.oracle.com/database/technologies/appdev/nodejs.html>
 - <https://oracle.github.io/node-oracledb/>
 - Installation: npm install oracledb
- Supports Oracle CRUD operations from JavaScript
- We'll modify our ProductDao implementation to work with Oracle tables



An Oracle-Based DAO Implementation

- Our DAO will now perform CRUD operations by calling `oracledb` functions
 - Note that all Oracle functions return Promises, so DAO methods are `async`
 - We add a `db_utils.js` module with utility functions for common database operations

```
const dbUtils = require('./db-utils');

class ProductDao {
  async queryForAllWidgets() {
    const sql = `select id, description, price, gears, sprockets
      from widgets order by id`;
    const widgets = [];

    const result = await dbUtils.getConnection().execute(sql, {}, dbUtils.executeOpts);

    const rs = result.resultSet;

    let row;
    while ((row = await rs.getRow())) {
      const newWidget = new Widget(row.ID, row.DESCRIPTION, row.PRICE, row.GEARS, row.SPROCKETS);
      widgets.push(newWidget);
    }
    await rs.close();
    return widgets;
  }
}
```

DAO method is `async`

Use `await` on DB calls

Import our module with database utility functions

Execute the SELECT

Get the values from a row of the result set

Add the new Widget to the array of Widgets

Return the array of Widgets

Code is similar to JDBC

Managing Database Transactions

- We need to add transaction management to our application
- DAO methods shouldn't manage transactions
 - DAO methods don't know when transactions should be committed or rolled back
 - Instead, the class that implements the use cases defines the transaction boundaries
 - In simple API implementations, the RESTful controller implements the use cases
 - In more complex applications, the methods of a business service define transactions
- In either case, we'll encapsulate transaction methods in a `TransactionManager` class

TransactionManager Implementation

- Our transaction manager is a Node.js version of the class we created in the JDBC course
 - All methods are `async` because they call `oracledb` asynchronous methods

```
class TransactionManager {  
  
  async startTransaction() {  
    await this.openConnection();  
  }  
  
  async openConnection() {  
    this.connection = await dbUtils.getConnection();  
  }  
  
  async commitTransaction() {  
    try {  
      await this.connection.commit();  
    }  
    finally {  
      await this.closeConnection()  
    }  
  }  
}
```

```
async rollbackTransaction() {  
  try {  
    await this.connection.rollback();  
  }  
  finally {  
    await this.closeConnection()  
  }  
}  
  
async closeConnection() {  
  try {  
    await dbUtils.closeConnection(this.connection);  
  }  
  catch (error) {  
    console.error(`database exception: ${error}`);  
    throw error;  
  }  
}
```

ProductDao Refactored

- TransactionManager and ProductDao need to use the same database connection
 - TransactionManager will own the connection
 - ProductDao will get the connection from TransactionManager
 - Controller passes the transaction manager to the DAO constructor

```
class ProductRestController {  
    constructor() {  
        this.transactionManager = new TransactionManager();  
        this.productDao = new ProductDao(this.transactionManager);  
    }  
}
```

TransactionManager will provide connections to ProductDao

```
class ProductDao {  
    constructor(connectionProvider) {  
        this.connectionProvider = connectionProvider;  
    }  
  
    async queryForAllWidgets() {  
        ...  
        const result = await this.connectionProvider.connection.execute(...);  
    }  
}
```

ProductDao accesses connection from TransactionManager

RESTful Controller with Asynchronous Methods

- The RESTful controller now has a DAO and a TransactionManager
 - Controller methods define the boundaries of database transactions

```
class ProductRestController {  
    constructor() {  
        this.transactionManager = new TransactionManager();  
        this.productDao = new ProductDao(this.transactionManager);  
    }  
  
    async addWidget(req, res) {  
        const w = req.body;  
        await this.transactionManager.startTransaction();  
        try {  
            const rowCount = await this.productDao.createWidget(w);  
  
            await this.transactionManager.commitTransaction();  
  
            res.json({ rowCount: rowCount });  
        } catch (err) {  
            await this.transactionManager.rollbackTransaction();  
            res.status(500).json({error: err});  
        }  
    }  
}
```

Methods use `await`, so they must be `async`

TransactionManager will provide connections to ProductDao

Call an `async` transaction manager method

Call an `async` DAO method

Smoke Test for Oracle Integration

1. Open a terminal and cd to
Node_JavaScript\RESTfulServices\Chapter8\NodeProducttApi
 - Run npm install and npm start
2. In SQL Developer, open the scott connection
 - Execute this SQL: select * from widgets;
3. Use Insomnia to send a GET request to
<http://localhost:3000/widgets>
 - Copy the JSON for one of the widgets from the response
4. Now send a POST request to <http://localhost:3000/widgets>
 - Set the request body type to JSON
 - Paste the widget JSON into the body and change id to 99
5. In SQL Developer, execute SELECT again and verify row was added
6. Use Insomnia to send a DELETE request to
<http://localhost:3000/widgets/99>
 - Use SQL Developer to confirm row was deleted

The screenshot shows the Insomnia REST Client interface. At the top, a green button says "POST" and the URL is "http://localhost:3000/widgets". Below the URL are tabs for "JSON", "Auth", "Query", "Headers", and "Docs". A purple "Send" button is on the right. The main area shows a JSON object with fields: "id": 99, "description": "Unbelievable Widget", "price": 99.99, "gears": 9, "sprockets": 9. This is followed by a "Beautify JSON" section. Below that is a summary bar with "200 OK", "74.8 ms", "14 B", and "2 Minutes Ago". Underneath is a "Preview" section with tabs for "Headers" (containing 7 items), "Cookies", and "Timeline". The preview shows a JSON response with a single field: "rowCount": 1.

Chapter Concepts

Callbacks and Promises

Accessing Oracle with Node.js

Testing Asynchronous Functions

Chapter Summary

Controller Integration Testing

- Now we can define integration tests for our RESTful controller
- Controller methods commit database changes
 - So, we'll need to re-initialize the Oracle database table before each spec
 - db_utils defines an executeDml() function that executes SQL DML statements
 - We'll call that function from beforeEach()

```
describe('specs for widget API', () => {
  beforeEach(async () => {
    const stmts = [
      "delete from widgets",
      "insert into widgets values (1, 'Low Impact Widget', 12.99, 2, 3)",
      "insert into widgets values (2, 'Medium Impact Widget', 42.99, 5, 5)",
      "insert into widgets values (3, 'High Impact Widget', 89.99, 10, 8)"
    ];
    await dbUtils.executeDml(stmts);
  })
});
```

Callback uses await,
so it must be async

Re-initialize the database

Controller Integration Testing (continued)

- Integration tests will use Axios to send HTTP requests to the deployed API
- Specs will call asynchronous database functions and controller methods
 - `it()` callbacks must be `async`

```
const baseUrl = 'http://localhost:3000/widgets';

const widgetsInDb = [
  {id: 1, description: 'Low Impact Widget', price: 12.99, gears: 2, sprockets: 3}, ...
]

describe("retrieve all widgets", () => {
  it("succeeds", async () => {
    const rowCount = await dbUtils.countRowsInTable('widgets');

    const response = await axios.get(baseUrl);

    expect(response.status).toBe(200);
    expect(response.data).toBeTruthy();
    expect(response.data.length).toEqual(rowCount);
    expect(response.data).toEqual(widgetsInDb);
  });
});
```

Get the initial state of the database

Send an HTTP GET request

Verify values in HTTP response

Writing a Negative Test Case

- As always, we need to include negative test cases in our test suite
- Potential negative test cases:
 - Invalid path parameters or query parameters for GET and DELETE requests
 - Invalid or missing data in the body of a POST or PUT request
- Axios functions throw errors if a response status is not in the 2xx range

```
it("fails due to an invalid ID", async () => {
  try {
    await axios.get(`${baseUrl}/-1`);
    fail('GET with invalid ID should have failed');
  }
  catch (err) {
    expect(err.response.status).toEqual(406);
    expect(err.response.data.error).toMatch(/widget.*id/i);
  }
});
```

Send a request with bad data

If Axios doesn't throw an error, make the spec fail

Verify error status

Use regular expressions (REs) to test error messages to minimize breaking changes

Spec for Adding a Widget

- For insert and update methods, call `db_utils.countRowsInTable()` to verify a row was added or modified correctly

```
it("succeeds", async () => {
  const oldRowCount = await dbUtils.countRowsInTable('widgets');
  const newRowId = oldRowCount + 1;
  const newWidget = { id: newRowId, description: 'Very Groovy Widget',
    price: 99.99, gears: 9, sprockets: 99 };

  const response = await axios.post(baseUrl, newWidget);
  expect(response.status).toBe(200);
  expect(response.data.rowCount).toEqual(1);

  const newRowCount = await dbUtils.countRowsInTable('widgets');
  expect(newRowCount).toEqual(oldRowCount + 1);

  const queryResult = await dbUtils.countRowsInTableWhere('widgets',
    `id = ${newRowId}
    and description='Very Groovy Widget'
    and price = 99.99
    and gears = 9
    and sprockets = 99`);
  expect(queryResult).toEqual(1);
});
```

Get the initial state of the database

Send HTTP POST to add a widget

Verify values in HTTP response

Verify new database state

Verify a row was added to the `widgets` table with all the correct values



HANDS-ON
EXERCISE

30 min

Exercise 8.2: Calling a Back-End RESTful API

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Callbacks and Promises

Accessing Oracle with Node.js

Testing Asynchronous Functions

Chapter Summary

Chapter Summary

After completing this chapter, you are now able to:

- Work with asynchronous JavaScript functions that return Promises
- Simplify asynchronous code with `async` and `await`
- Test asynchronous RESTful methods using the Jasmine framework
- Perform CRUD operations on an Oracle database using Node.js

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 9: The FidZulu Mini Project

Chapter Overview

In this chapter, we will explore:

- The FidZulu mini project

Chapter Concepts

The FidZulu Mini Project

Chapter Summary

Introduction to the FidZulu Mini Project

■ Background:

- Fidelity wants to develop a service to compete with Amazon
 - For this purpose, you will develop a Single Page Application using node and Angular only

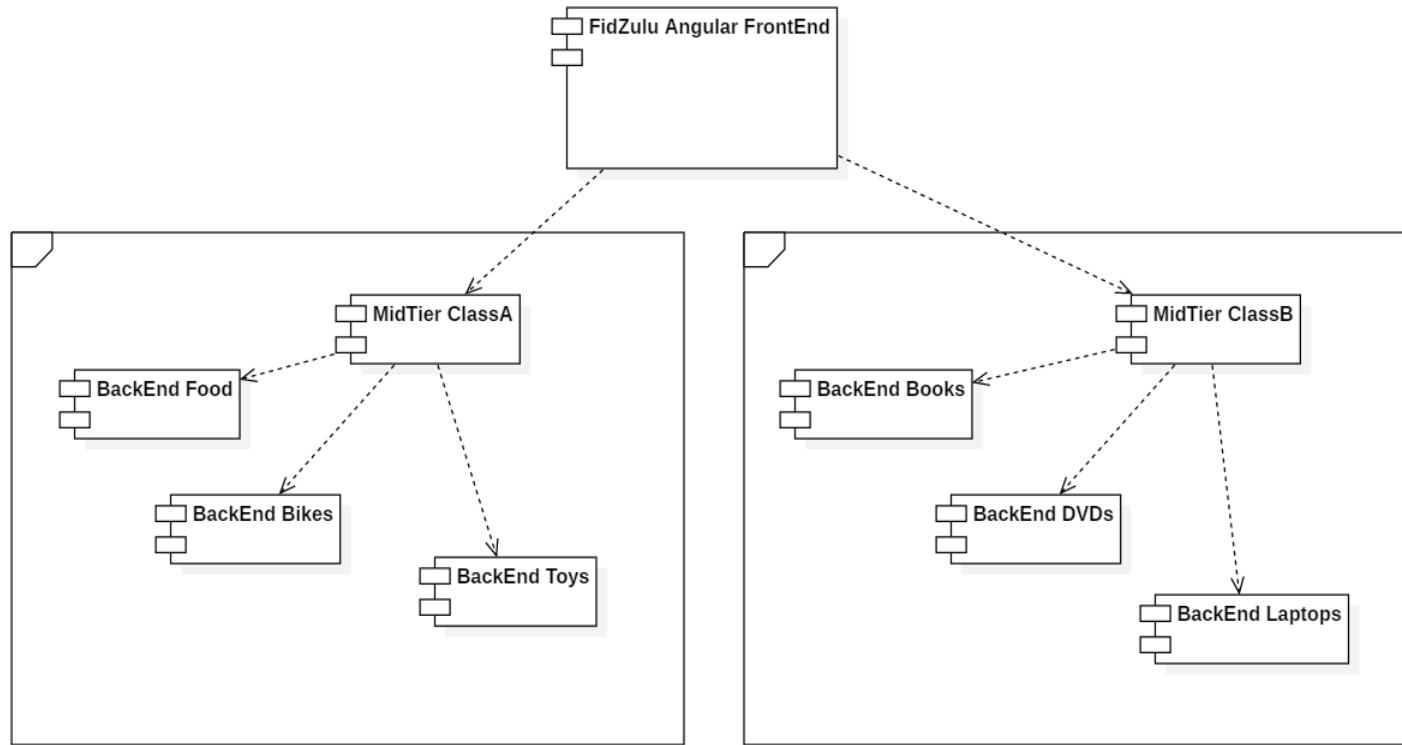
■ Scope:

- For the first phase of this project, you will develop six RESTful services using Node.js and Express
 - Each service will be responsible for a certain data type
- There will also be a RESTful service that will manage two of the data services and provide a composite API for the underlying functionality
- For the front-end, an Angular application will be developed accessing the composite RESTful services to receive the necessary data

■ For complete details, refer to the FidZulu project document

- Your instructor will provide the location of this document

The FidZulu Architecture



Chapter Concepts

The FidZulu Mini Project

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The FidZulu mini project

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 10: Apigee Edge

Chapter Overview

In this chapter, we will explore:

- What Apigee Edge is
- What types of problems Apigee Edge solves

Chapter Concepts

Apigee Edge

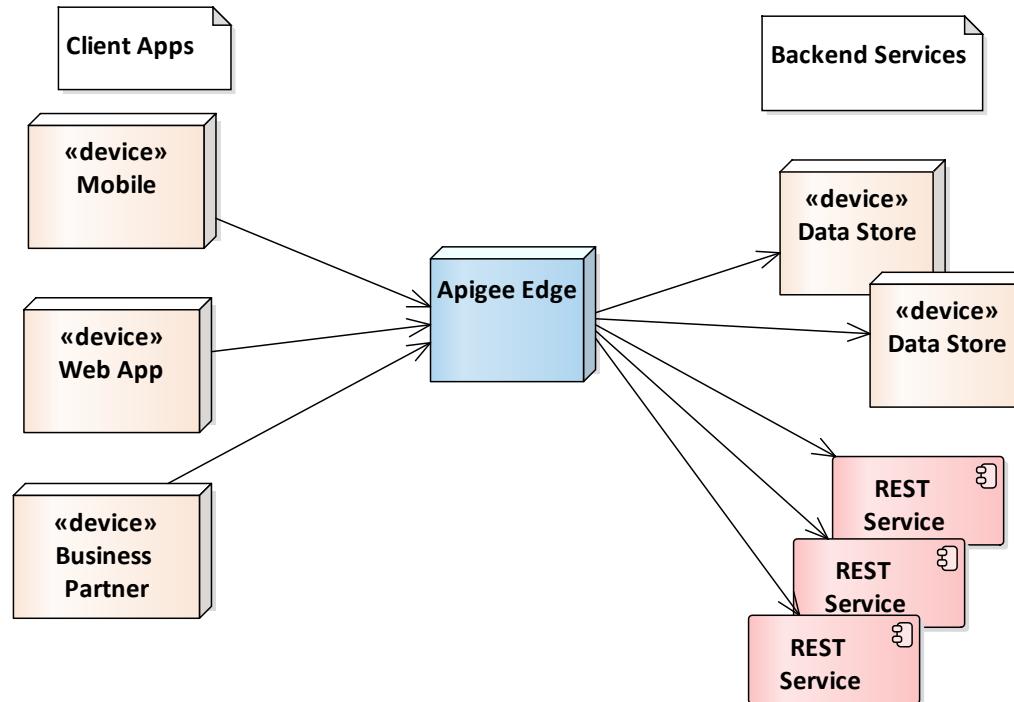
Chapter Summary

Understanding Apigee Edge

- Apigee Edge is a platform for developing and managing API proxies
- An API proxy is a proxy for a back-end service that provides other features
 - Security
 - Rate limiting
 - Quotas
 - Analytics
- Front-end developers will communicate with the API proxies in order to access the back-end services

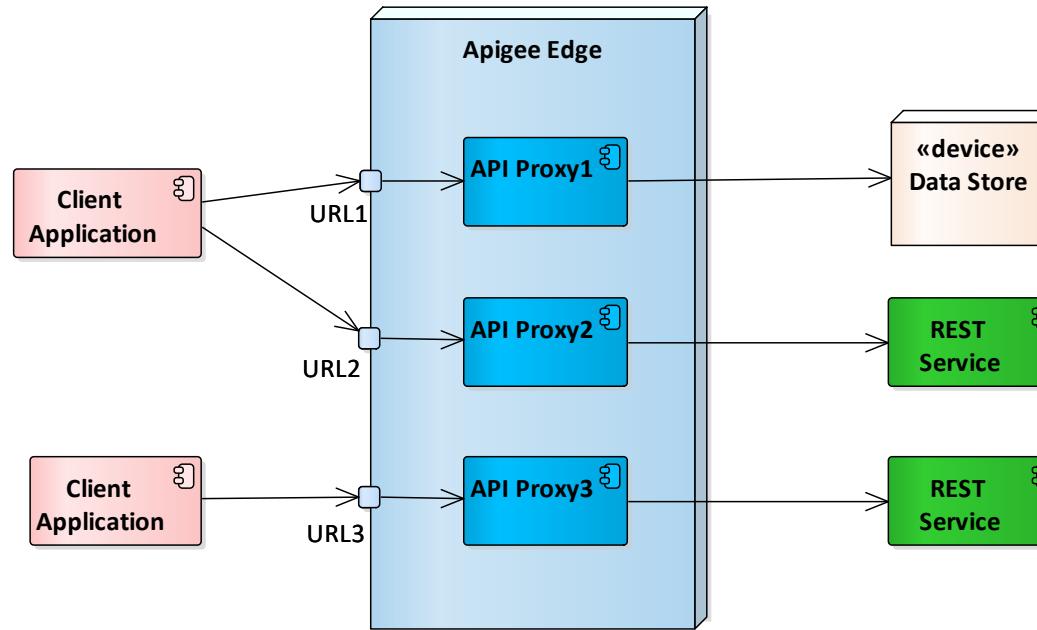
Enterprise Architecture with Apigee Edge

- This diagram illustrates how Apigee Edge can be used in an enterprise application



Client Communication

- Client applications communicate with an API proxy to access back-end resources



API Proxies

- The API proxy isolates the client developer from back-end resource details
 - Client developers do not need to know details of resource implementations
- Client developers need to know:
 - The URL of the API proxy endpoint
 - Parameters or headers passed in a request
 - Authentication and authorization credentials
 - Format of the response

API Key

- Client developers must register the application with Apigee
 - The client developer will receive an API key
- Every client request must include the API key in every request to an API proxy
- Keys can be revoked at any time
 - Clients using that key will no longer have access to the services
- Keys can also have a time limit placed on them
 - The key must be refreshed when the time expires

Controlling API Proxies

- Flows are used to control how an API proxy performs
 - Define business logic
 - Add conditional behavior
 - Determine error handling
- Flows are sequential steps that define a processing path
- Policies can be added to flows to implement various features
 - Security such as OAuth
 - Traffic management
 - Message manipulation such as returning data in XML or JSON
 - Caching data across requests
 - CORS support

API Proxy Design and Development

- Apigee provides guidelines, documentation, and examples for working with Apigee Edge
- Getting started with Apigee Edge
 - <https://docs.apigee.com/api-platform/get-started/get-started>
- Best practices for API proxy design
 - <https://docs.apigee.com/api-platform/fundamentals/best-practices-api-proxy-design-and-development>
- Apigee Edge antipatterns
 - <https://community.apigee.com/articles/44662/the-book-of-apigee-edge-antipatterns.html>

Chapter Concepts

Apigee Edge

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- What Apigee Edge is
- What types of problems Apigee Edge solves

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 11: Service Virtualization

Chapter Overview

In this chapter, we will explore:

- What service virtualization is
- What types of problems service virtualization solves
- How to use virtual services in enterprise development

Chapter Concepts

Service Virtualization

Chapter Summary

Service Virtualization

- A major development bottleneck is waiting on dependent components
- Service virtualization allows the use of virtual services instead of production services
 - Enables development before a dependent component is available
 - Emulates the behavior of a dependent component
 - Allows integration testing much sooner

Mocking vs. Service Virtualization

- Mocks are fake software components used to imitate real software components
 - Tend to be very context specific
 - Simulate a specific response to a certain request
 - Usually simulates an individual class
 - Best suited for unit tests
- Virtual services can be deployed throughout the entire production cycle
 - Consistently delivering functionality that developers and testers can use
 - Eliminate the need for individual developers or testers to write and rewrite mocks
 - Can simulate an entire network of backend services
 - Best suited for integration and performance tests
- Both approaches have value, and both will often be used in application development

Use Cases for Service Virtualization

- Consuming a third-party web service such as a credit check
 - Use service virtualization to avoid paying for web service calls
- Test web service response with certain data
 - Use service virtualization to create the desired response
- Automated tests expect valid and invalid data
 - Service virtualization can provide both types of data
- Testing front-end web clients
 - Virtual services can run many tests at one time on a single server
- Enterprise application with dozens of service dependencies
 - Simulate the back-end service dependencies with virtual services

How Does Service Virtualization Work?

- Service virtualization software typically uses one of two methods to generate virtual components
- Generate virtual service code from a standard service description
 - Requires some setup and refinement
 - Each virtual service only needs to be created once
 - Then can be used repeatedly and continuously in development and testing
- Create traffic recordings of actual system interactions between your application and the dependent components
 - Potentially simpler approach
 - Does require the dependent component to be available for recording

Service Virtualization Tools

There are many software tools that support service virtualization:

- Hoverfly
- ServiceV Pro
- CA LISA
- Micro Focus Service Virtualization
- IBM Green Hat
- Tricentis TOSCA Orchestrated Service Virtualization
- Soap UI
- Parasoft Virtualize
- Traffic Parrot for Microservices

Hoverfly

- Hoverfly is an open-source API (service) simulation tool
 - Lightweight
 - High performance
 - REST API
 - Extend and customize with any programming language
- Extensive documentation is available here:
<https://hoverfly.readthedocs.io/en/latest/index.html>



HANDS-ON
EXERCISE

20 min

Exercise 11.1: Using Service Virtualization

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Service Virtualization

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- What service virtualization is
- What types of problems service virtualization solves
- How to use virtual services in enterprise development

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 12: Testing with Cucumber.js

Chapter Overview

In this chapter, we will explore:

- The basics of regular expressions
- Behavior-Driven Development (BDD) with Cucumber

Chapter Concepts

Acceptance Testing

Regular Expressions

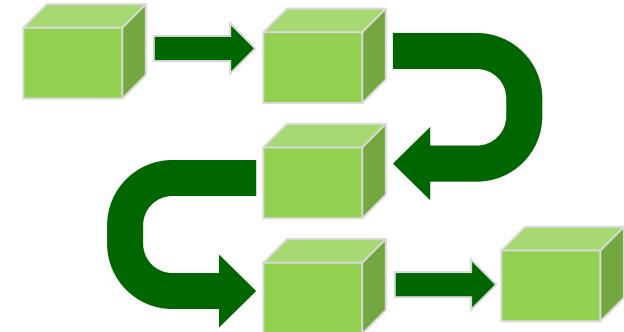
Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

End-to-End Testing

- End-to-end testing (E2E)
 - A methodology used to test the flow of an application
 - From start to finish
- Testing the application from the user point of view
 - Treats the application as a blackbox
 - Only the user interface is exposed to the user (tester)
- Usually performed after functional and system testing



End-to-End Testing

Types of End-to-End Testing

- See <https://static1.smartbear.co/smartbear/media/images/landingpages/testcomplete/end-to-end-graphic.png>
- Horizontal: from the user's perspective
 - Occurs horizontally across the context of multiple applications
 - This method can easily occur in single ERP (Enterprise Resource Planning) application
 - Take an example of a web-based application of an online ordering system
 - The whole process will include accounts, inventory status of the products, as well as shipping details
- Vertical: verify each layer of the application's architecture from top to bottom
 - All interactions with an application are verified
 - From start to finish
 - Each layer of the application is tested from top to bottom
 - Much more thorough and much harder to implement

Simple Horizontal End-To-End Testing Scenario: ATM

- A simple test flow for an everyday task: withdrawing cash from an ATM
 - Customer inserts card
 - ATM prompts for PIN
 - Customer enters valid PIN
 - ATM prompts for operation
 - Customer selects Withdraw
 - ATM prompts for amount
 - Customer enters \$100
 - ATM dispenses \$100
 - ATM prompts if customer wants receipt
 - Customer selects Yes
 - ATM dispenses receipt
 - ATM prompts if customer wants another transaction
 - Customer selects No
 - ATM returns customer's card
- Scenario is written in simple, non-technical language
- Scenario is written from the user's point of view
 - Doesn't include system behavior that isn't visible to the user

Types of E2E Test Cases

- Tests should be designed from the user's point of view
- Tests should focus on testing some existing features of the application
- Multiple scenarios should be considered
- Different sets of test cases should focus on different scenarios of the application

Designing End-to-End Testing

■ User functions

- Listing features of the software systems
- Document actions performed, input and output
- Document relationships between different actions

■ Conditions

- For each user function, document its conditions
- Timing, data conditions, etc.

■ Test cases

- For each scenario, create one or more tests cases
- Each condition should be covered in a separate test case

Automated Acceptance Testing: Cucumber

- Cucumber: tool for running automated E2E **acceptance tests**
 - Tests are written in a Behavior-Driven Development (BDD) style
 - Focuses on using examples to talk through how an application behaves
- Cucumber introduced the **Gherkin** language for outlining acceptance test scenarios
- Input to Cucumber is a **feature file** written in business-facing text
- Features describe user goals:

As a *<user/role>* I want to *<goal>* So that *<business value>*

- Scenarios define acceptance criteria:

Given *pre-conditions*

When *action or event that initiates the scenario*

Then *desired result or outcome*

Example: Withdraw Cash from ATM

- Features and scenarios are written in simple, non-technical language
 - All stakeholders on a project can understand the application's requirements

■ Feature:

As a bank customer

I want to withdraw cash from my account at an ATM

So I can make purchases

■ Scenario:

Given My ATM card is valid

And The ATM dispenser contains cash

And My account has funds

When I request a \$100 cash withdrawal

Then My account is debited by \$100

And \$100 cash is dispensed

And A receipt is dispensed

And My card is returned

Advantages of Using Cucumber

- Cucumber takes this scenario exactly as written and generates skeleton test code
- Provides a direct path from:
 1. User story (As a...I want to...So...), to
 2. Gherkin scenario description (Given...When...Then...), to
 3. Cucumber automated acceptance test case, to
 4. Code that makes the test case pass
- Every stakeholder can read and understand the features and scenarios
 - Users, business analysts, project sponsors, testers, developers
- Feature files become the “single source of truth” for project requirements

Example: CEO of Fidelity

- We can also use parameters in the scenarios to make our tests more flexible
 - Quoted strings become parameters in test functions generated by Cucumber

Feature:

As a Fidelity FSE
I want to search the Web
So I get fast, accurate answers to questions
to help me complete tasks quickly

Scenario: Google search for CEO of Fidelity

Test function parameter

Given I have an internet connection available
When I search Google for "CEO of Fidelity Investments"
Then I should see "Abigail Johnson" in the result



HANDS-ON
EXERCISE

10 min

Exercise 12.1: Writing an Acceptance Test with Gherkin Syntax

- Write a feature file for making a purchase at Amazon
- Please refer to your Exercise Manual to complete this exercise
- Discuss with instructor your approach before you get started

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

Regular Expression

- To access parameters in some feature files, we need *Regular Expressions* (RE)
- What is a Regular Expression?
 - As sequence of characters, meta-characters, operators, and precedence rules which describe a set of strings
 - Useful for sophisticated pattern matching
- Use REs to capture parameters in scenarios in your feature files
- To explore REs, browse to <https://regex101.com/>

Basic Regular Expression Metacharacters

Regular Expression	Set of Strings			
a	a			
ab	ab			
a [abc]	aa ab ac			
a [a-c]	aa ab ac			
a [] a-c]	a] aa ab ac			
a [] a-c-]	a] aa ab ac a-			
a [^a-z]	a followed by not a through z			

Basic Regular Expression Metacharacters (continued)

Regular Expression	Set of Strings
a.	a followed by any character
a\.	a.
a\\	a\

Basic Regular Expression Metacharacters (continued)

Regular Expression	Set of Strings
a*	zero or more of the letter a
aa*	a followed by zero or more of letter a
. *	The * repeats any character zero or more times
[a-c] *	This matches any string of characters zero or more a, b, or c in any order

Do Now



- For each set of strings, write an RE which matches the set of strings

1. sherlock

2. Any string of at least three characters

3. 1.00 2.00 3.00 4.00 5.00 6.00

Do Now (continued)



4. The string "bob" followed by any number of lowercase characters

5. The string .\-[]

Anchors: Definition

- In UNIX, regular expressions are always compared against lines
- A line is a sequence of zero or more characters terminated, but not including, a new line
- Anchors position regular expressions within the line

Anchors: Definition (continued)

Anchor	Explanation
^the	the at the beginning of a line (i.e., column 1)
the\$	the at the end of a line
^the\$	the at the beginning of line and end of line (i.e., the are the only characters on the line)
\<the	the at the beginning of a word (i.e., a word boundary: start of line, after a space, after punctuation)
the\>	the at the end of a word (i.e., end of line, before a space, before punctuation)
\<the\>	The word the (not inside a longer word)

Anchors: Definition (continued)

- **Note:** \< and \> are not supported by JavaScript or on regex101.com
 - Use \b instead

Anchor	Explanation
\bthe	The \b matches the at a word boundary
\Bthe\B	The \B matches the empty string NOT at the edge of a word followed by the (i.e., the is inside a word)

More RE Metacharacters

- Most programming languages support additional RE metacharacters

RE	Explanation
\w \W	A word character (a-z A-Z _), a non-word character
\d \D	A digit (0-9), a non-digit
\s \S	A whitespace character, a non-whitespace character
{ 3 }	Exactly three occurrences of the preceding RE
{ 0 , 3 }	Zero through three occurrences of the preceding RE
?	Zero or one occurrence of the preceding RE
+	One or more occurrences of the preceding RE

Reading Regular Expressions

- Reading a complex RE can be very challenging
 - regex101.com gives a detailed explanation of an RE
 - Example: an RE that matches a US postal code

The screenshot shows the regex101.com interface. On the left, under 'REGULAR EXPRESSION', is the pattern `^\\d{5}(-\\d{4})?$/gm`. Below it, under 'TEST STRING', is a list of nine lines:
12345
1234
123456
1234a
a2345
12345-6789
12345-678
1234-67890

The first four lines (12345, 1234, 123456, 1234a) do not match the RE. The last five lines (a2345, 12345-6789, 12345-678, 1234-67890) are highlighted in green, indicating they are matches.

On the right, under 'EXPLANATION', is a detailed breakdown of the RE:

- ^ asserts position at start of a line
- \d matches a digit (equivalent to [0-9])
- {5} matches the previous token exactly 5 times
- 1st Capturing Group (-\d{4})?
 - ? matches the previous token between zero and one times, as many times as possible, giving back as needed (greedy)
 - matches the character - with index 45₁₀ (2D₁₆ or 55₈) literally (case sensitive)
 - \d matches a digit (equivalent to [0-9])
 - {4} matches the previous token exactly 4 times
- \$ asserts position at the end of a line

Do Now



■ What will each of the following regular expressions check for?

1. ab^*c

2. The

3. testing

4. $\bb{[ou]}y\bb$

Creating Regular Expressions

1. Write down the list of the strings to be matched
2. If there are anchors, write them down and start with a character before or after the anchor
3. Look at the first character of each string to be matched
 - Write down an RE to describe this letter
4. Look at the next few letters
 - Are they repeats of the last character written down?
 - If so, add an RE to specify these characters
5. Go to Step 3 and use the next character as the first character



Do Now: Regular Expressions

- Write a regular expression that will match a floating-point number without a leading sign
-

- Write a regular expression which can handle a telephone number with format (999) 999-9999
-

- Check your regular expression at <https://regex101.com/>

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

BDD: Putting It All Together

- Let's try to make this all work in the real world
- We'll create a BDD project to test a simple calculator class
- We need the following steps to run our test:
 - Create the necessary directory structure
 - Install necessary Node.js libraries
 - Create your feature file
 - Run Cucumber to generate skeletons for your specs
 - Write your `*-steps.js` spec file and implement functionality for Gherkin language

Cucumber Expression

- Instead of REs, you can access a scenario parameter with a Cucumber Expression (CE)
- CEs are a newer Cucumber feature that are simpler than REs and easier to learn
- CE syntax: `{param-type}`
 - `{string}` – captures a double-quoted string from the feature scenario
 - `{int}` – captures an unquoted number from the feature scenario

```
Given('I can connect to SimpleService', () => { ... });
```

Given I can connect to SimpleService
When I make a request for all "contacts"
Then I should receive 2 contacts

```
When('I make a request for all {string}', (items) => { ... });
```

Assigns string from feature scenario to `items` parameter

```
Then('I should receive {int} contacts', (count) => { ... });
```

Assigns number from feature scenario to `count` parameter



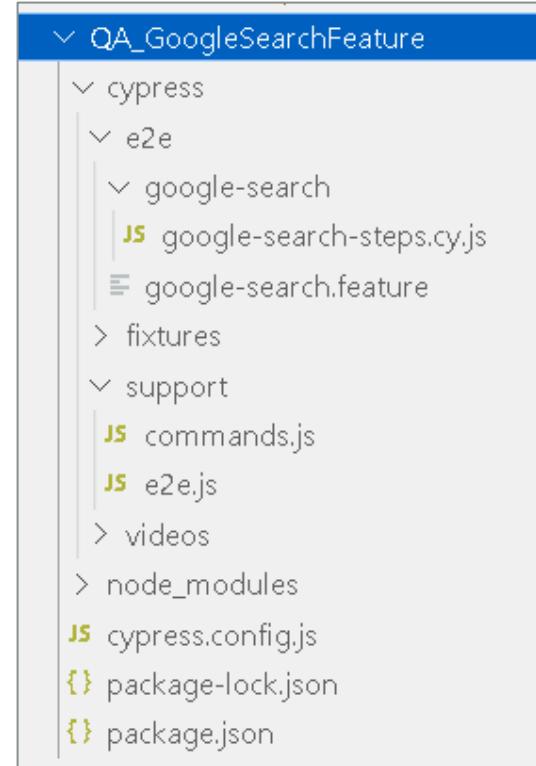
Exercise 12.2: Test Your Calculator

20 min

- Explore the steps file and feature file in this example
- Make the tests work for the calculator project
- Please refer to your Exercise Manual to complete this exercise

Structure for Cucumber Project with Cypress

- By integrating Cucumber with Cypress, we can automate tests of a RESTful API
- Project structure:
 - The `cypress/e2e` folder stores all feature files
 - For every `.feature` file, create a folder with the same name to store all related `steps.cy.js` files
 - See at right: `google-search`
 - `cypress.config.js` and `cypress/support/e2e.js` contain configuration



The Feature File

- Write feature descriptions and scenarios in a text file
 - Filename extension .feature
- Install VS Code extension:
Cucumber
 - Smart editor for feature files
 - Enables autocompletion, syntax highlighting
- Enter the feature description and one or more scenarios

```
search.feature > ...
Feature: Google Search Testing
As a Fidelity FSE
I want to use the Google search page
So I get accurate answers to questions
and I can complete tasks quickly and efficiently

Scenario: Google search for CEO of Fidelity
Given I have internet available
When I search Google for "CEO of Fidelity Investments"
Then I should see "Abigail Johnson" in the search results

Scenario: Google search for the product of 7 and 6
Given I have internet available
When I search Google for "7 * 6"
Then I should see "42" in the search results
```

Coding Your Steps File

- The project's package.json file will define dependencies for Cypress and Cucumber
- In your step definition file, load the necessary Cucumber functions:

```
import { Given, When, Then } from "cypress-cucumber-preprocessor/steps";
```

- Call the Cucumber Given() function:

```
const url = 'https://www.google.com';
```

Argument to Given() is the text
from the feature description

```
Given('I have internet available', () => {
  cy.request(url).its('status').should('equal', 200);
})
```

Given I have internet available

When I search Google for "CEO of Fidelity Investments"

Then I should see "Abigail Johnson" in the search results

Coding Your Steps File (continued)

```
When I search Google for "CEO of Fidelity Investments"
Then I should see "Abigail Johnson" in the search results
```

Using Cucumber
Expressions instead of REs

Quoted string in scenario is
passed as `text` parameter

■ Call When()

```
When('I search Google for {string}', (text) => {
    // access google page
    cy.visit(url);

    cy.get('input[name="q"]').type(text + '{enter}');
})
```

`type()` simulates typing search
text into Google input field

■ Call Then()

```
Then('I should see {string} in the result', (text) => {
    cy.get('a').should('contain', text);
});
```

■ Test case passes if assertions in Given(), When(), and Then() are all true



20 min

Demo 12.3: Who Is the CEO?

- This an instructor-led demonstration
- You can explore the code for this exercise, however, due to fast changing versions for Chrome browser, your version might not execute properly
- Please refer to your Exercise Manual for demo details

Chapter Concepts

Acceptance Testing

Regular Expressions

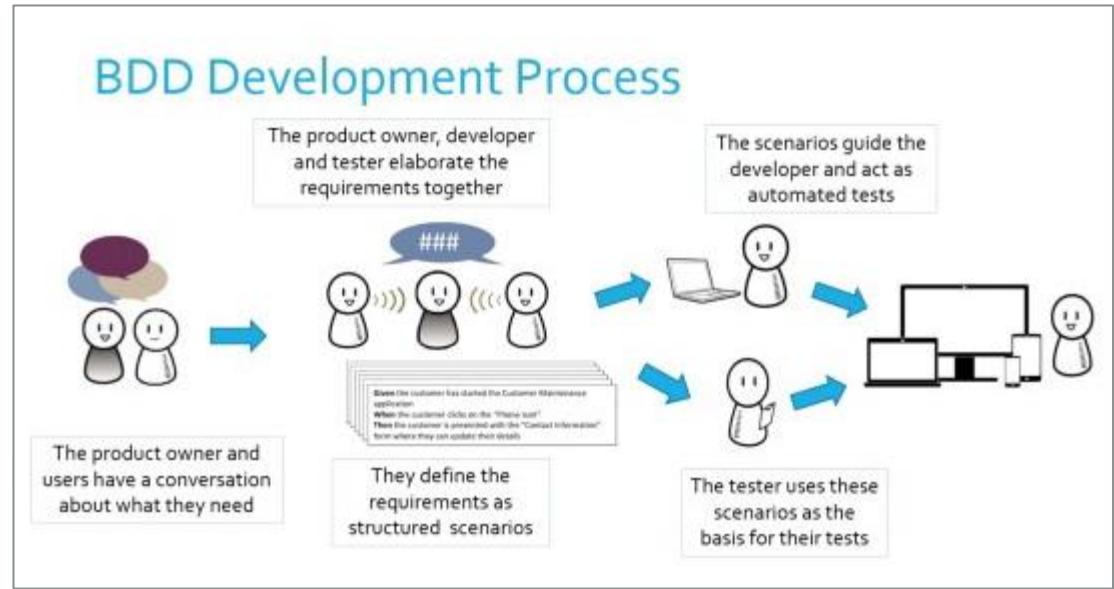
Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

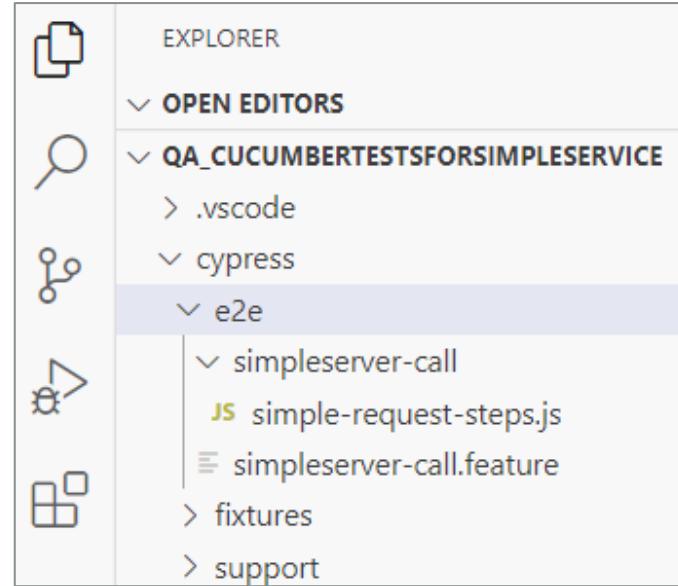
BDD: How to Test RESTful Services

- The previous example needed selenium support (or any similar web driver) to handle browser actions
- RESTful Services don't have a web interface, so how does our scenario differ?
- Thus, we need a slightly different set of node modules, but the principles remain the same



Simplified Structure for this Cucumber Project

- The e2e folder is used to store all feature files and step files
 - Store feature files directly under the e2e folder
 - Inside the e2e folder, create a folder to store all steps.js files



Feature File for RESTful Service

- We'll use Cucumber and Cypress to test a RESTful Service
- Feature file:

```
Feature: Accessing SimpleService
  As a Fidelity FSE
  I want to retrieve contacts from SimpleService
  So I can contact my team members

  Scenario: Getting contacts from SimpleService
    Given I can connect to SimpleService
    When I make a request for all "contacts"
    Then I should receive 2 results
```

Coding Your Steps File

- First, load the necessary libraries and define global variables

```
simpleservice-steps.cy.js
import { expect } from "chai";
import { Given, When, Then } from "cypress-cucumber-preprocessor/steps";
```

```
let responseBody;
const url = 'http://localhost:3000';
```

simpleservice.feature

```
Given I can connect to SimpleService
When I make a request for all "contacts"
Then I should receive 2 results
```

- Given

```
// Scenario 1
Given('I can connect to SimpleService', () => {
  cy.request('GET', url).its('status').should('equal', 200);
  // Given() can be used for authentication or other configuration
})
```

Coding Your Steps File (continued)

```
Given I can connect to SimpleService  
When I make a request for all "contacts"  
Then I should receive 2 results
```

When

```
When('I make a request for all {string}', (searchText) => {  
    cy.request(`${url}/${searchText}`).should((res) => {  
        responseBody = res.body; // extract response body  
    });  
})
```

Using Cucumber Expressions

Then

```
Then('I should receive {int} results', (count) => {  
    expect(responseBody.result.length).to.equal(count);  
  
    // We must use the "expect" assertion here,  
    // because "should" works only with cy() return values  
});
```

Running Cypress in Headless Mode

- We don't need a browser to test a RESTful API
- Can run Cypress in headless mode
 - Test results written to console
 - Useful in automated builds
- Modify test script in package.json
 - Replace cypress open with cypress run
- Run npm test as usual

```
/d/Node_JavaScript/RESTfulServicesSolutions/Chapter12/QA_CucumberTestsForSimpleService $ npm test  
> cucumber-test@1.0.0 test  
> cypress run  
  
Running: simpleserver-call.feature (1 of 1)  
Accessing SimpleService Server  
  ✓ Getting contacts from SimpleService (207ms)  
  ✓ Getting single contact from SimpleService (90ms)  
  ✓ Getting status 404 when contact does not exist (64ms)  
3 passing (435ms)  
\(Results\)  
  
Tests:      3  
Passing:    3  
Failing:    0  
Spec Ran:  simpleserver-call.feature  
=====  
\(Run Finished\)  
  


| Spec                        | Tests | Passing | Failing | Pending | Skipped |
|-----------------------------|-------|---------|---------|---------|---------|
| ✓ simpleserver-call.feature | 429ms | 3       | 3       | -       | -       |
| ✓ All specs passed!         | 429ms | 3       | 3       | -       | -       |


```



Exercise 12.4: Let's Test SimpleService BDD Style

30 min

- Program a Cucumber project to test SimpleService
- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The basics of regular expressions
- Behavior-Driven Development (BDD) with Cucumber

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 13: Server-Side JavaScript Programming

Chapter Overview

In this chapter, we will explore JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators
- Using Grunt

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Understanding 'this'

- In JavaScript, the thing called **this** is a reference to the object that “owns” the executing JavaScript code
- When used in a global function:
 - **this** is Node’s `global` object (equivalent to browser’s `window` object)
- When used in a method of an object (that is, a function bound to an object):
 - **this** is the object itself
- When used in an object constructor:
 - **this** is the object under construction
- When used in an arrow function:
 - **this** is unchanged (that is, it’s the same as the arrow function’s enclosing scope)

Using call(), apply(), and bind()

- JavaScript functions have methods
 - Including `call()`, `apply()`, and `bind()`
- The `call()` and `apply()` methods can be invoked immediately
- However, the `bind()` method returns a bound function that, when executed later, will have a chosen context for `this` when calling the original function
- **Note:** none of these functions can be used with arrow functions
 - Because `this` is bound to scope at initialization

Using call()

- Either `call()` or `Function.prototype.call()` can be used
- Allows us to explain what `this` is in a function without context for `this`
- The first parameter in the `call` method provides the context for `this`

```
// Demo with javascript .call()
let obj = { name: "Fidelity" };

function greeting(a, b, c) {
    return "Welcome to " + this.name + " where " + a + " is " + b + " and " + c;
};

let obj = greeting.call(obj, "working", "fun", "rewarding");
// "Welcome to Fidelity where working is fun and rewarding"
```

Using apply()

- Either `apply()` or `Function.prototype.apply()` can be used
- Similar to `call()`, `apply()` allows to identify `this` in a function without context
- However, in this case we have two parameters:
 - The bound object
 - A list of arguments, which is passed to the called function

```
//Demo with javascript .apply()
let obj = { name: "Fidelity" };

function greeting(a, b, c) {
    return "Welcome to " + this.name + " where " + a + " is " + b + " and " + c;
};

let args = ["working", "fun", "rewarding"];
let msg = greeting.apply(obj, args);

// "Welcome to Fidelity where working is fun and rewarding"
```

Using bind()

- In this case, we are binding the object without invoking the function
 - We get an altered function back
 - Now we can call the new function to get the correct response

```
//Demo with javascript .call()
let obj = { name: "Fidelity" };

function greeting(a, b, c) {
    return "Welcome to " + this.name + " where " + a + " is " + b + " and " + c;
};

let boundFunctionRef = greeting.bind(obj);

console.dir(boundFunctionRef); // returns [Function: bound greeting]

console.log(boundFunctionRef("working", "fun", "rewarding"));

// "Welcome to Fidelity where working is fun and rewarding"
```

Immediately Invoked Functions

- JavaScript supports Immediately Invoked Function Expressions (IIFE)
 - Pronounced “iffy”
- A function definition is the “normal” way of creating a named function
- You can assign a function expression to a variable or property
- If we want to evaluate the function right away (like immediately):
 - Just add the parentheses at the end
- Can pass arguments too

```
function normalNamedFunction() { /*...*/ }
normalNamedFunction(); // call the function
```

```
var varFunc = function () { /* ... */ };
varFunc(); // call the function
```

```
(function () {/* ... */})(); // call the IFFE
```

```
var msg = "Howdy";
(function (funcArg) {
  console.log(funcArg);
})(msg);
```

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Object Literal

- As you already know, an object literal is a fast way to create an object with defined values
 - *Note:* object literals cannot be further instantiated, only use them for singletons!

```
var cell = {  
    name : "LG",  
    model : "Stylo",  
    weight : 144.6,  
    color : "silver",  
    fullname : function() {  
        return this.name + " " + this.model;  
    }  
};
```

- Values can still be changed later, but no other objects of that type can be created

```
cell.name = "Samsung";  
cell.model = "Galaxy S7";
```

Function Literal (or Function Expression)

- A function literal is very similar to a constructor function, but is unnamed
 - It allows parameters and multiple instances

```
var cell = function (name, model, weight, color) {  
    var name = name; // made name private  
    var model = model; // made model private  
    this.weight = weight; // these are public  
    this.color = color;  
    this.changeColor = function(color) {  
        this.color = color;  
    }  
}  
  
var myLG = new cell("LG", "Stylo", 144.6, "silver");  
myLG.changeColor("red");
```

- However, function expressions cannot be hoisted
 - They only exist when code is reached
- What is hoisting?

Hoisting

- Hoisting means that I can call a function before it is defined in the code sequence

```
// Output: "Hello!"  
functionTwo();  
  
function functionTwo() {  
    console.log("Hello!");  
};
```

```
// TypeError: undefined is not a function  
functionOne();  
  
var functionOne = function() {  
    console.log("Hello!");  
};
```

Class Definitions

- New to ECMAScript 6 are classes and inheritance
 - Similarities to TypeScript should make adaptation easier

Before ECMAScript 6

```
var Shape = function (id, x, y) {
    this.id = id;
    this.move(x, y);
};

Shape.prototype.move = function (x, y) {
    this.x = x;
    this.y = y;
};
```

```
class Shape {
    constructor(id, x, y) {
        this.id = id;
        this.move(x, y);
    }
    move(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

Inheritance

Before ECMAScript 6

```
var Rectangle = function (id, x, y, width, height) {  
    Shape.call(this, id, x, y);  
    this.width = width;  
    this.height = height;  
};  
  
Rectangle.prototype = Object.create(Shape.prototype);  
Rectangle.prototype.constructor = Rectangle;  
  
var Circle = function (id, x, y, radius) {  
    Shape.call(this, id, x, y);  
    this.radius = radius;  
};  
  
Circle.prototype = Object.create(Shape.prototype);  
Circle.prototype.constructor = Circle;
```

ECMAScript 6 version

```
class Rectangle extends Shape {  
    constructor(id, x, y, width, height) {  
        super(id, x, y);  
        this.width = width;  
        this.height = height;  
    }  
}  
  
class Circle extends Shape {  
    constructor(id, x, y, radius) {  
        super(id, x, y);  
        this.radius = radius;  
    }  
}
```

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

What Is a Factory Function?

- The purpose of the object factory is to create objects
- Usually implemented in a class or a static method of a class
 - Can produce repeatedly similar objects
 - Provides a way to users of the factory to create objects without knowing the specific type (class) at compile time
- Objects created by the factory method are by design inheriting from the same parent object
 - However, there are specific subclasses implementing specialized functionality
 - Sometimes the common parent is the same class that contains the factory method
- Though it is possible to use some ECMAScript 6 features, most of the technology used here was available before

What Should the Factory Do for Us?

- We want to have a method that accepts a type given as a string at runtime and then creates and returns specific objects of that type
- We don't want to use a constructor ("new" statement) to create these objects
 - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```

Let's Build Cars

- We create a common parent `CarMaker` constructor
- Then we add a static method of the `CarMaker` called `factory()`, which creates car objects
- And also add a `prototype.drive` function to provide that feature to all cars

```
// parent constructor
function CarMaker() {}

// a method of the parent
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};
```

Courtesy: JavaScript Patterns by Stoyan Stefanov

Let's Build Cars (continued)

- Now we need the specific car models

```
CarMaker.Compact = function() {  
    this.doors = 2;  
};  
  
CarMaker.Convertible = function() {  
    this.doors = 4;  
};  
  
CarMaker.SUV = function() {  
    this.doors = 17;  
};
```

Let's Build Cars (continued)

```
// the static factory method
CarMaker.factory = function (type) {
    var newcar;

    // error if the constructor does not exist
    if (typeof CarMaker[type] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }
    // at this point the constructor is known to exist
    // let's have it inherit the parent but only once
    if (typeof CarMaker[type].prototype.drive !== "function") {
        CarMaker[type].prototype = new CarMaker();
    }
    // create a new instance
    newcar = new CarMaker[type]();
    // optionally call some methods and then return.....
    return newcar;
};
```

Let's Build Cars (continued)

- We now have a method that accepts a type given as a string at runtime and then creates and returns objects of that type
- There is no constructor used with new
 - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```



HANDS-ON
EXERCISE

30 min

Exercise 13.1: Factories

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Understanding Scope

- In JavaScript, scope is the set of variables that the code currently has access to
 - The set of variables, objects, and functions that can be accessed
- JavaScript has lexical scoping
 - With function scope
 - Even though it looks like it should have block scope ({ ... })
- A new scope is created only when a new function is created
- Nested functions
 - The inner function has access to the outer function scope
 - Known as “lexical scope”
 - Aka “closure”

Nested Functions

- Nested functions have access to outer function scope

```
function GangOfFour() {  
  let amigo1 = 'Grady';  
  
  function scope1() {  
    console.log(`The first amigo: ${amigo1}`);  
  
    function scope2() {  
      let amigo2 = 'Ivar';  
      console.log(`Two amigos: ${amigo1} ${amigo2}`);  
  
      function scope3() {  
        let amigo3 = 'James';  
        console.log(`Three egos: ${amigo1} ${amigo2} ${amigo3}`);  
      };  
      scope3();  
    };  
    scope2();  
  };  
  scope1();  
};  
  
GangOfFour();
```

Output:

```
The first amigo: Grady  
Two amigos: Grady Ivar  
Three egos: Grady Ivar James
```

Block Scope with ECMAScript 6

- Block Scoping was introduced 2015 with ECMAScript 6

```
let callbacks = [];
for (let i = 0; i <= 2; i++) {
    callbacks[i] = () => i * 2;
}
console.log(callbacks[0]() === 0); //true
console.log(callbacks[1]() === 2); //true
console.log(callbacks[2]() === 4); //true
```

callbacks
array is global

"i" only exists
within block

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Closures Step by Step

- A closure is a function having access to the parent scope
 - Even after the parent function has closed (that is, returned)
- Let's go step by step to understand why we need closures

The Scope Problem

- We already know private variables and global variables

```
function myFunction() {  
    var a = 4;  
    return a * a;  
}
```

```
var a = 4;  
function myFunction() {  
    return a * a;  
}
```

- However, the lifetime of these variables is very different due to their nature
 - Private variables live within the function they are declared in; whenever the function is called, the variable is created
 - Global variables live as long as your window/web page does
 - A variable that is created **without** the **var** key word is ALWAYS GLOBAL

The Counter Problem

No privacy!

```
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();

// the counter is now equal to 3
```

Too private

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();

// want the counter to be 3
// but it does not work
```

Correct result, but everyone can overwrite counter, without add()

Every time we call add(), the counter is newly created and set to 1

Nested Functions

```
function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();  
    return counter;  
}
```

- This could work, if we could reach the plus function
 - So far, we can only reach that function if we create an object from add
 - But that is NOT what we are looking for
- If only we could find a way to execute **var counter = 0** only once

Closures

- Remember IIFE functions and what they do?
- What is happening here?
- The first `add` (declared with `var`) invokes the entire function
 - `add` receives the inner function as return value
- When you call `add()` you actually only invoke the inner function without recreating the counter variable
- This is a **closure** which allows functions to have private variables

```
var add = (function () {  
  var counter = 0;  
  return function () {  
    return counter += 1;  
  }  
})();  
  
console.log("counter = " + add());  
console.log("counter = " + add());  
console.log("counter = " + add());
```

Objects vs. Closures

Objects	Closures
Can add functions later; flexibility	Cannot add function; safety
Can use function from other source; reuse of code!	Has to create function always from scratch; more memory used; possible redundancy
Need to be careful when a method gets detached from an object, <code>this</code> will get a different meaning; complexity!	You don't need to keep track of <code>this</code>
WHEN TO USE	
Less concern with privacy, many instances of an object required	High privacy requirements and few "objects" of that type are needed



HANDS-ON
EXERCISE

20 min

Exercise 13.2: Exploring Closures

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Processing Collections

- Processing a collection of items is very common
- JavaScript supports several ways of iterating over a collection
 - Simple `for` loops
 - Iterators and generators
 - Provide a mechanism for customizing the behavior of `for...of` loops

The for Loop

- There are two forms of the `for` loop:

- `for...of` – loops over the items of an array
- `for...in` – loops over the keys (properties) of an object

```
stats = ['Rimac', 'Nevera', 256, 2_100_000]

for (let value of stats) {
  console.log(` ${value}`);
}
```

Output:

```
Rimac
Nevera
256
2100000
```

```
let vehicle = { make: 'Rimac',
  model: 'Nevera',
  topSpeedMph: 256,
  priceUsd: 2_100_000 };

for (let key in vehicle) {
  console.log(` ${key}: ${vehicle[key]}`);
}
```

Output:

```
make: Rimac
model: Nevera
topSpeedMph: 256
priceUsd: 2100000
```

Iterators

- An iterator knows how to access the items in a collection
 - One at a time
 - Keeps track of its current position in the collection

- JavaScript iterators

- Provide a `next()` function
 - Returns the next item in the collection
 - The object returned has two properties
 - `done`
 - `value`

```
function makeAnIterator(array) {  
    var nextIndex = 0;  
  
    return {  
        next: function() {  
            return nextIndex < array.length ?  
                {value: array[nextIndex++], done: false}  
                : {done: true};  
        }  
    };  
}
```

```
var it = makeAnIterator(['Grady', 'Ivar', 'James']);  
console.log(it.next().value); // 'Grady'  
console.log(it.next().value); // 'Ivar'  
console.log(it.next().done); // false
```

Iterables

- An iterable in ES6 is an object that defines its iterator
- The `for...of` loop can loop over any iterable
- You can create your own iterables
 - Define a function on the object names
`@@iterator`
 - Or use `Symbol.iterator` as the function name
- Since JavaScript does not have interfaces:
 - Iterable is a convention
- JavaScript does provide some built-in iterables
 - String
 - Array
 - Map
 - Set

```
let iterableUser = {  
  name: 'Grady',  
  lastName: 'Booch',  
  [Symbol.iterator]: function* () {  
    yield this.name;  
    yield this.lastName;  
  }  
}  
  
// logs 'Grady' and 'Booch'  
for(let item of iterableUser) {  
  console.log(item);  
}
```

Generators

- A generator is a special function
 - Allows you to write an algorithm that maintains its own state
 - And can be paused and resumed
- A generator is a factory for iterators
- A generator function is marked with an *
 - And contains at least one yield statement

```
function* generateRandomNumbers(){  
  let start = 1;  
  let end = 42;  
  while(true)  
    yield Math.floor((Math.random() * end) + start);  
}
```

```
//no execution here  
//just getting a generator  
let sequence = generateRandomNumbers();  
  
for(let i=0; i<5; i++){  
  console.log(sequence.next());  
}
```

Advanced Generators

- Generators compute their yielded values on demand
 - Efficiently represent a sequence of values that are expensive to compute
 - Or even an infinite sequence!
- The `next()` function accepts an input argument
 - Can be used to modify the internal state of the generator
 - Will be used as the result of the last `yield` expression of the generator
- Note: generators do NOT like recursion!

```
function* fibonacci() {  
  var fn1 = 1;  
  var fn2 = 1;  
  while (true) {  
    var current = fn1;  
    fn1 = fn2;  
    fn2 = current + fn1;  
    var reset = yield current;  
    if (reset) {  
      fn1 = 1;  
      fn2 = 1;  
    }  
  }}}
```

```
var sequence = fibonacci();  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 2  
console.log(sequence.next().value); // 3  
console.log(sequence.next(true).value); // 1  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 2  
console.log(sequence.next().value); // 3
```



HANDS-ON
EXERCISE

20 min

Optional Exercise 13.3: Iterators and Generators

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

What Is Grunt?

- Grunt is a task-based command line build tool for managing JavaScript projects
- What kind of tasks?
 - Running tests
 - Minifying code
 - Running JSHint on your code
- Grunt is built on Node.js
 - Available via the Node Package Manager (npm)
 - Installs a bunch of dependencies

```
npm install -g grunt
```

Using Grunt

- Grunt needs a `grunt.js` project file to run
- The '`grunt init`' command will create the `grunt.js` file
 - Specify what type of project type you want
 - `jquery`: a jQuery plugin
 - `node`: a Node module
 - `commonjs`: a CommonJS module
 - `gruntplugin`: a Grunt plug-in
 - `gruntfile`: a Gruntfile (`grunt.js`)
- You will need to provide some values
 - Project name
 - Project title
 - Etc.

Some Grunt Commands

■ Some of the things Grunt will do for you

- grunt lint – checks your JavaScript using JSHint
- grunt qunit – runs your Qunit tests
- grunt concat – concatenates project files together and places the new file in the `dist` folder
- grunt min – minifies the file produced by the `concat` command

Customizing Grunt

- You can customize Grunt
 - Edit the `grunt.js` file
- Properties of the config object passed to Grunt
 - `pkg` – points to the `package.json` file which stores project metadata
 - `meta` – object with only a `banner` property which is placed at the top of the concatenated (or minified) file
 - `concat / min / qunit / watch` – set options for each task (such as the files to operate on)
 - `JSHint` – examines your JavaScript code for possible problems
- The end of the file defines the default task
 - `grunt.register('default', 'lint qunit concat min')`

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Chapter Summary

In this chapter, we have explored JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators
- Using Grunt

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 14: Functional and Reactive Programming in JavaScript

Chapter Overview

In this chapter, we will explore:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

Reactive Programming

Chapter Summary

The `forEach` () Function

- To iterate over a collection of objects:
 - Can use a `for` loop
 - Or use the `forEach` function
- The `forEach` function is defined in `Array.prototype`
 - Requires a function that will execute for each object in the array

```
var products = [  
  { name: 'Golf clubs', price: 175.0 },  
  { name: 'Basketball', price: 25.0 },  
  { name: 'Tennis racket', price: 95.0 },  
  { name: 'Baseball glove', price: 65.0 },  
  { name: 'Catnip mouse', price: 5.5 }  
];  
  
// traditional for loop  
for (product of products) {  
  console.log(product);  
}  
  
// the Array.forEach function  
products.forEach(product => console.log(product));
```

Which to Use?

- The `forEach` function improves readability
 - The next object is automatically passed to the function argument
 - Don't have to use loop counter variables
- Fewer off-by-one errors with `forEach`
 - No loop counter variables
 - No off-by-one bug
- The `for` loop allows breaking out early
 - If you need to break out of the loop before iterating completely through it:
 - Use the `break` key word
 - Not available in `forEach`

Usage of "break"

```
let i;
for (i = 0; i < products.length; i++) {
  if (products[i].name == "Tennis racket") {
    break;
  }
}
console.log('Tennis racket price is ' +
  products[i].price);
```

The map () Function

- The `map()` function is defined in `Array.prototype`
- Creates a new array
 - Containing the results of calling the provided function on each array element
- The function passed to `map()` can take three arguments
 - The current object
 - The index of the current object
 - The array being processed
- The `map()` function can take an optional second argument
 - The object to be used as `this`

Global functions and array

```
function isOdd(num) {  
    return num % 2 !== 0  
}  
function timesTwo(num) {  
    return num * 2  
}  
let numbers = [1, 4, 9];
```

```
let doubles = numbers.map(timesTwo);  
// or  
let doubles = numbers.map(num => num * 2);  
// doubles is now [2, 8, 18]  
// numbers is still [1, 4, 9]
```

The filter() Function

- What if you only want to transform some of the values?
 - Map works on all the values
- The filter() function solves this problem
 - It takes a function that returns a Boolean
 - True, keep the value
 - False, discard the value

```
function isOdd(num) {  
  return num % 2 !== 0  
}  
function timesTwo(num) {  
  return num * 2  
}
```

```
let numbers = [1, 2, 3, 4];  
let newNumbers = numbers.filter(isOdd) // [ 1, 3 ]  
                      .map(timesTwo) // [ 2, 6 ]  
  
// or using arrow functions  
let newNumbers = numbers.filter(n => number % 2 !== 0)  
                      .map(n => n * 2);
```

The `reduce()` Function

- What if I want to sum the values in an array?
 - Using the `reduce()` function does the trick
- The first argument to `reduce()` is a callback function
 - `reduce()` will call this function for each array item
 - Callback's first parameter is the value returned by the previous call
 - Callback's second parameter is an array item
- The second argument to `reduce()` is the initial value of the callback's first parameter

```
var numbers = [1, 2, 3, 4];  
  
var totalNumber = numbers.map(number => number * 2)  
                    .reduce((total, nextNumber) => total + nextNumber, 0);  
  
console.log("The total number is", totalNumber); // 20
```

Initial value
of total

Function called for
each array item



HANDS-ON
EXERCISE

20 min

Exercise 14.1: Working with Arrays

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

Reactive Programming

Chapter Summary

Higher Order Functions

- A higher order function is a function that takes another function as an argument
 - Like map, filter, reduce, etc.
 - The argument function is often referred to as a callback
- A function that returns a function:
 - Is also known as a higher order function

No More Loops

- Using higher order functions
 - There is no need for most imperative loops
 - No need to use the for loop
- The map, filter, reduce functions
 - Apply an argument function to every element in a collection
- Side-effect-free
 - Array higher order functions do not mutate the array they are called on



20 min

Optional Exercise 14.2: Using Higher Order Functions

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

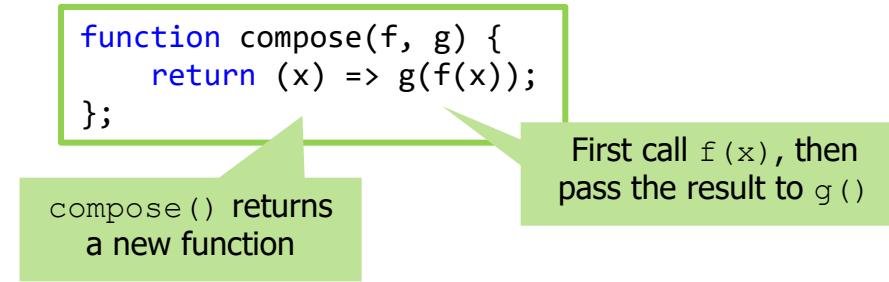
Reactive Programming

Chapter Summary

Function Composition

- Function composition is the process of combining two or more functions
 - To produce a new function:

`compose(f, g)(x)` equals `g(f(x))`



- Function order is similar to piping in *nix

Function Composition (continued)

- What will the following produce?

```
function trim(str) {  
    return str.replace(/^\s*|\s$/g, '');  
}  
  
function capitalize(str) {  
    return str.substring(0,1).toUpperCase() +  
        str.substring(1);  
}  
  
let convertFunc = compose(trim, capitalize);  
  
let convertedStr = convertFunc(' abc def ghi ');  
console.log(`"${convertedStr}"`);
```

```
function compose(f, g) {  
    return (x) => g(f(x));  
};
```

```
function add1(x) {  
    return x + 1;  
}  
  
function mult2 (x) {  
    return x * 2;  
}  
  
// what will this produce?  
let f = compose(add1, mult2);  
console.log(f(7));
```



HANDS-ON
EXERCISE

20 min

Exercise 14.3: Using Function Composition

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

Reactive Programming

Chapter Summary

Currying

- Currying is an application of function composition
 - Constructs functions that allow partial application of the function's arguments
 - If you pass all the args in a call, you get the result
 - If you pass a subset of the args, you get a function that expects the rest of the args
 - The function caches the subset of args in a closure for later use

```
function multiply(x, y) {  
    return x * y;  
}  
  
function curriedMultiply(x) {  
    return (y) => x * y;  
}
```

Returns a function that multiplies its parameter by x

x is cached in a closure

```
multiply(5,2) === multiplyBy5(2);
```

```
var multiplyBy5 = curriedMultiply(5);
```

Currying and Callbacks

- Currying can be combined with callbacks
 - To create a higher order *factory* function
- Useful in event handling
- Can replace the callback pattern used in Node.js
- This code is an example of combining currying with Node.js
 - To process a file
 - Allows for the read data to be passed around
 - As the file is being processed
 - Defer invoking the read function's callback
 - Until the result is needed
 - Can allow for sequential and parallel i/o processing of multiple files

```
// a curried version of node's fs.readFile(path,
encoding, callback) function
var readfileC = curriedReadfile(path, encoding);

readfileC(function(err, data) {
  if (err) {
    throw err;
  }
  // do something clever with the data
});
```

The Power of Currying

- With currying, you can separate the initiation of an asynchronous operation
 - From the retrieval of the result
- So, it is possible to initiate several operations in close sequence
 - Let them do their I/O in parallel
 - Retrieve the results later

```
var reader1 = curriedReadFile(path1, "utf8");
var reader2 = curriedReadFile(path2, "utf8");
// I/O is parallelized and we can do other important things while it runs

// further down the line:
reader1(function(err, data1) {
  reader2(function(err, data2) {
    // do something clever with data1 and data2
  });
});
```



Optional Exercise 14.4: Currying

20 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

Reactive Programming

Chapter Summary

Reactive Programming

■ Reactive systems are:

- Responsive
 - Responds in a timely manner
 - Focus on providing rapid and consistent response times
- Resilient
 - Stays responsive in the face of failure
- Elastic
 - Stays responsive under varying workload
- Message-driven
 - Relies upon asynchronous message passing

From *The Reactive Manifesto*, <http://www.reactivemanifesto.org/>

From Promises to Observables

- Promises act on data
 - And then return a single value
- Observables are the observer pattern at work
 - Can produce multiple values asynchronously over time
- An observable is a function that takes an observer
 - And returns a cancellation function
- An observer is an object with next, error, and complete functions

The Rx Library

■ Reactive Extensions (RxJs)

- A library for composing asynchronous applications
- Using observable sequences and LINQ-style query operators
- Works with synchronous and asynchronous data streams

	Single Return Value	Multiple Return Values
Pull/Synchronous/Interactive	Object	Iterables(Array...)
Push/Asynchronous/Reactive	Promise	Observable

LINQ – Language Integrated Query

Reactive Programming in JavaScript

- Reactive programming revolves around asynchronous data
 - Called observables
 - Or streams
- There are quite a few choices for frameworks that extend JavaScript
 - And add support for reactive programming
 - Reactive Extension (RxJs)
 - ReactiveX
 - Omniscient
 - WebRx

Observable Example

- The basic building blocks of RxJs
 - Observables (producers)
 - Observers (consumers)
- Two types of observables
 - Hot observables
 - Pushing even if we are not subscribed to them
 - Cold observables
 - Pushing only when we subscribe to them

```
// Creates an observable sequence of 5
// integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Prints out each item
var subscription = source.subscribe(
  function (x) { console.log('onNext: %s', x); },
  function (e) { console.log('onError: %s', e); },
  function () { console.log('onCompleted'); });

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

Prime Number Generator Example

- Suppose we want to aggregate the results from a prime number generator over time
 - So, the user interface does not have to deal with too many updates
 - We are interested in only the number of generated prime numbers
- Probably want to use a buffer
 - RxJs provides a buffer function
- Also, may want to use a map
 - To transform the data
- The `fromEvent` function constructs an observable

```
var worker = new Worker('prime.js');
var observable = Rx.Observable.fromEvent(worker, 'message')
    .map(function (ev) { return ev.data * 1; })
    .buffer(Rx.Observable.interval(500))
    .where(function (x) { return x.length > 0; })
    .map(function (x) { return x.length; });
```



HANDS-ON
EXERCISE

20 min

Exercise 14.5: Using Observables

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Composition

Currying

Reactive Programming

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

Resources

- There is much more to the testing JavaScript
- Here are some resources to aid in further explorations



- **JavaScript: The Good Parts**
 - Douglas Crockford
 - <https://www.youtube.com/watch?v=hQVTIJBZook>
 - Available on Safari Books Online



- *Test-Driven JavaScript Development*
 - Gupta, Prajapati & Singh, Packt Publishing
 - Available on Safari Books Online

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Course Summary

Course Summary

In this course, we have:

- Solved common programming problems by using design patterns
- Designed and built RESTful web services
- Used Spring Boot to create RESTful web services written in Java
- Used Node.js to create RESTful services written in JavaScript
- Used advanced JavaScript programming techniques

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Appendix A: Building RESTful Services with JAX-RS

Appendix Overview

In this appendix, we will explore:

- What a RESTful web service is
- Building RESTful services with JAX-RS
- Testing RESTful web services
- Spring Support for JAX-RS

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

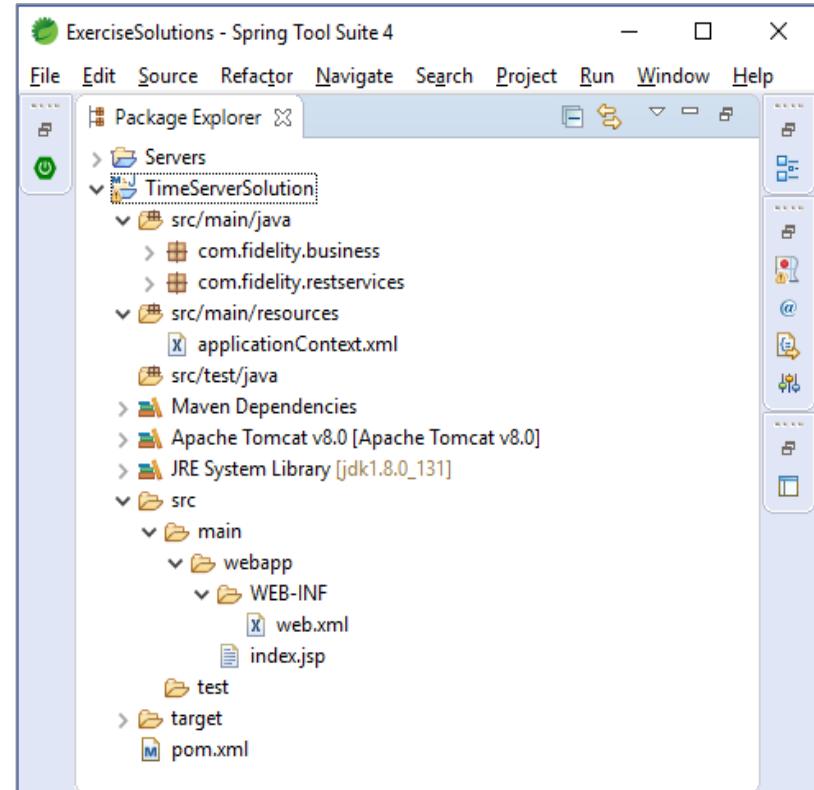
Structure of Maven Web Project

Java Resources

- src/main/java
 - Java classes
- src/main/resources
 - Resources like Spring configuration files

Deployed Resources

- src/main/webapp
 - The deployed web application
- WEB-INF/web.xml
 - A deployment descriptor used for servlet mappings and many other tasks
 - Any file under WEB-INF can be used by the server, but will not be available as a URL



Using web.xml to Map URLs

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>JaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.fidelity.restservices</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
  </servlet-mapping>
</web-app>
```



HANDS-ON
EXERCISE

30 min

Exercise A.1: Exploring the Time Service

- Please refer to your Exercise Manual to complete this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

What Is JAX-RS?

- JAX-RS is a standard Java API
 - Specification is JSR311 (Java Service Request)
 - Reference implementation is called Jersey
 - Other implementations: Apache's CXF, JBoss' RESTEasy

<http://cxf.apache.org/>

<https://resteasy.github.io/>

Writing a Service

■ Steps to write a service:

- Write a plain Java class with methods
 - Use annotations to mark it as a service
 - Use annotations to mark methods as web service methods
- For each web service method in the class, specify:
 - URL of operation
 - HTTP operation
 - MIME types produced/consumed
 - Path, query, or form parameters used as input
- Deploy web archive

Step 1: ServletContainer Configuration

- In web.xml, configure ServletContainer servlet: specify packages and URL pattern

```
<web-app ... >
  <servlet>
    <servlet-name>JaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.fidelity.greeter</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
  </servlet-mapping>
</web-app>
```

ServletContainer looks for RESTful services in this list of packages

ServletContainer intercepts URLs that start with jaxrs;
e.g., http://server/context/jaxrs/greet

Step 2: Write Service

- The service class is a plain Java class
 - Has to have a `@Path` annotation but its value could be an empty string
 - This is the URL pattern corresponding to this class

```
@Path("/greet") // @Path("") is also allowed
public class GreeterService {

}
```

- Because of the URL specified in `web.xml`, the full URL for this class is:

`http://localhost:8080/greeterService/jaxrs/greet`

- The machine URL comes from a Domain Name Service (DNS)
- The application context (`greeterService`) comes from name of WAR file
- `jaxrs` comes from `web.xml` URL pattern
- `greet` comes from the `@Path` of the service class

Step 3: Write Method

- For each method, specify the HTTP operation, URL, MIME types produced/consumed
 - Return values will be coerced into the MIME type specified in a common-sense manner

```
@GET  
@Path("/languages")  
@Produces(MediaType.TEXT_HTML) // or @Produces("text/html")  
public String getSupportedLanguages(){  
    return "<html><body><ol><li>English</li><li>French</li></ol></body></html>";  
}
```

- Note that the `@Path` adds to the path of the class, so that above method is invoked when user browses to the following URL:

```
http://localhost:8080/greeterService/jaxrs/greet/languages
```

Handling Form Inputs

- Can have form inputs injected as parameters to web service method

```
<form action="jaxrs/greet" method="POST">
    Your name:
    <input type="text" name="fullName"></input>
    <select name="language">
        <option value="fr">French</option>
        <option value="en">English</option>
    </select>
    <input type="submit" value="Submit" />
</form>
```

HTML form (client of web service)

- The web service method looks like this:

```
@POST
// no special path, so gets path of class
@Produces(MediaType.APPLICATION_JSON)
public String greetForm(@FormParam("fullName") String name,
                       @FormParam("language") String lang){
    // use name, lang normally
}
```

Raising Error Conditions

- Can raise HTTP error conditions by throwing a WebApplicationException

```
public String greetForm(@FormParam("fullName") String name,
                      @FormParam("language") String lang){
    String greeting;
    if ( "fr".equals(lang) ){
        greeting = "Bonjour, ";
    } else if ("en".equals(lang)){
        greeting = "Good morning, ";
    } else {
        throw new WebApplicationException(Status.BAD_REQUEST);
    }
    return "<greeting>" + greeting + name + "</greeting>";
}
```

- Can also have method return a Response object which contains only a HTTP Status (no document)

Handling Path Parameters

- Can inject variables in the URL to a method:

```
@GET  
@Path("/{language}/{name}")  
@Produces(MediaType.APPLICATION_JSON)  
public String greetPath(@PathParam("name") String name,  
                        @PathParam("language") String lang){  
    // use name, lang as normal  
}
```

- Example of Path handled by the above method:

```
http://localhost:8080/greeterService/jaxrs/greet/fr/Jane%20Doe
```

- Similarly, @CookieParam, @QueryParam, etc.

Query Parameters

- Query parameters can also be injected:

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public String greetPath(  
    @DefaultValue("Guest") @QueryParam("name") String name,  
    @DefaultValue("en") @QueryParam("lang") String lang){  
        // use name, lang as normal  
}
```

- Examples of URLs handled by the above method:

http://localhost:8080/greeterService/jaxrs/greet?lang=fr&name=Jane%20Doe

http://localhost:8080/greeterService/jaxrs/greet?name=Jane%20Doe

http://localhost:8080/greeterService/jaxrs/greet

http://localhost:8080/greeterService/jaxrs/greet?lang=fr

Allowed Parameter Types

- Path and Query parameters can be used on:

- String
- All primitive types (and their wrappers) except `char`
- Any class with the static method `valueOf(String)`
 - So, any enum works
- Any class that has a constructor that takes only one `String`
- `List<T>`, `Set<T>`, or `SortedSet<T>` where `T` matches above criteria

```
@GET  
@Path("/details/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public String showDetails(@PathParam("id") Employee employee){  
    // etc.  
}
```

```
public class Employee {  
    public static Employee valueOf(String id){  
        // etc.  
    }  
}
```

XML and JSON

- JSON is the standard format used with REST services
- Clients are often Dynamic Web Applications using JavaScript
- Jax-RS handles both JSON and XML data
 - Will marshal data to/from Java classes based on type of data
- If a client requires data from a service in a particular format, they should set the HTTP header on the request
 - Accept: application/json or Accept:application/xml
 - Service will automatically send data in the correct format
 - With no change to code
- For a service to accept data, client should set HTTP header indicating type of data being sent
 - Content-type: application/json or Content-type:application/xml

Producing JSON or XML Example

- Consider the following service:

- Will produce XML or JSON based on HTTP header and data format

```
@Path("/exhibits")
@Singleton
public class ExhibitsService {
    private List<Exhibit> exhibits = new ArrayList<>();

    public ExhibitsService(){
        // create initial exhibits for service and add to list above
    }
    @Path("/all")
    GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Exhibit> getExhibits(){
        return exhibits;
    } ...
```

Producing JSON or XML Example (continued)

- Following method will accept JSON or XML based on header set:

```
@Path("/add")
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public void addExhibit(Exhibit exhibit){
    exhibits.add(exhibit);
}
```

```
@XmlRootElement
public class Exhibit {
    private String name;
    private String artist;
```

```
// get/set methods
```

```
}
```

Exhibit parameter will be created from
JSON or XML data



HANDS-ON
EXERCISE

30 min

Exercise A.2: Creating a RESTful Service

- Please refer to your Exercise Manual to complete this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Testing RESTful Services as a POJO

- It is possible to test a RESTful service as a POJO using JUnit
- Simply create an instance of the service using a constructor
- In the test methods, call the service methods that you want to test
- This will verify that the service works locally

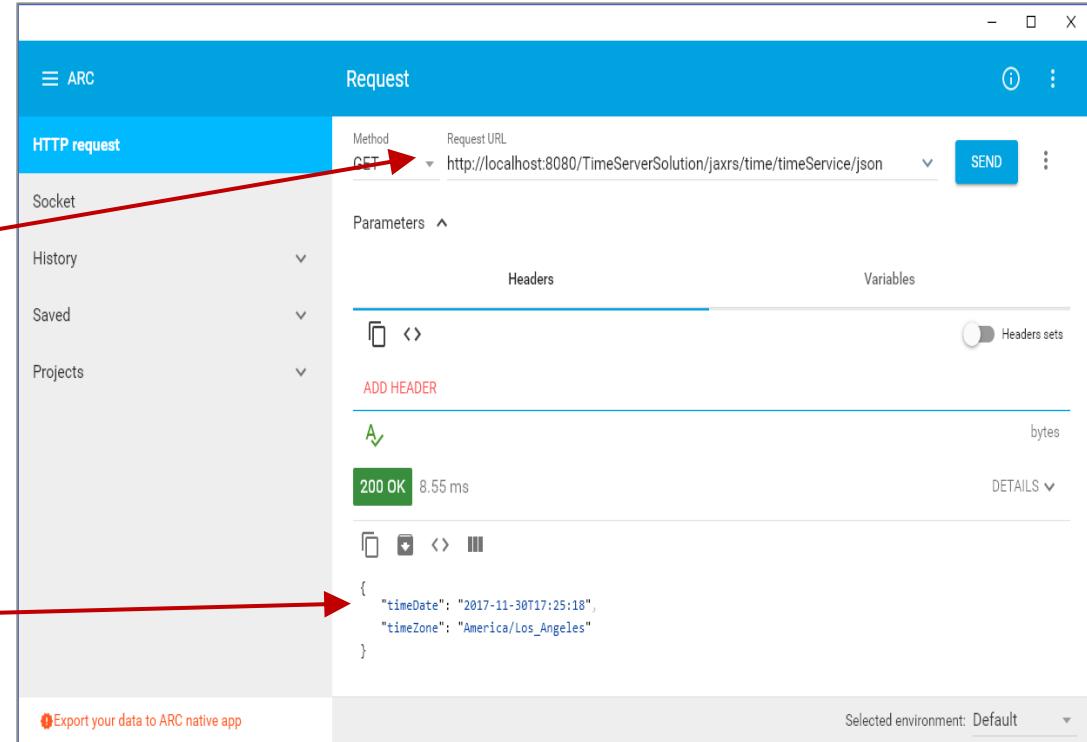
```
public class RestTimeServiceTest {  
    RestTimeService service;  
  
    @Before  
    public void setUp() throws Exception {  
        service = new RestTimeService();  
    }  
  
    @Test  
    public void testgetTime() {  
        TimeZoneInfo info = service.getTime();  
        assertNotNull(info);  
    }  
}
```

Testing RESTful Services as a Service

■ It is possible to test a RESTful service as a Service using tools like the Advanced Rest Client (ARC)

■ Using the tool, enter the URL for the service method that you want to call

- You can set parameters, pass data in the body of the request, and set HTTP headers
- The response data will be displayed once the request is sent





Exercise A.3: Testing a RESTful Service

40 min

- Please refer to your Exercise Manual to complete this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Spring Support for RESTful Services

- The RESTful services we have created so far have a problem
- They are created by the ServletContainer servlet
 - Which means they are not Spring managed beans
 - Which means we cannot ask Spring to inject any dependencies into our RESTful services
- But wait! There is good news!
 - Spring provides an answer to this problem
 - Spring provides a servlet context listener
 - When the servlet is initialized, the listener creates a Spring context (object factory)
 - When the servlet is destroyed, the listener closes the Spring context

Configuring Spring Support for RESTful Services

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:beans.xml</param-value>
</context-param>

<servlet>
    <servlet-name>SpringJaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>com.fidelity.restservices</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>SpringJaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
</servlet-mapping>
```

Define Spring context listener and beans.xml configuration file

These sections are unchanged (normal servlet container)



Exercise A.4: Integrating Spring with JAX-RS

30 min

- Please refer to your Exercise Manual to complete this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Appendix Summary

In this appendix, we have explored:

- What a RESTful web service is
- Building RESTful services with JAX-RS
- Testing RESTful web services
- Spring Support for JAX-RS